Internet Information Server Auxiliary Processor For Dyalog APL/W

ISAP

Version 5.4

Programmer's Reference

Lingo Allegro USA, Inc.

1105 Chicago Avenue
Suite 155
Oak Park, IL 60302
USA

Phone:	+1-800-LINGO-A1-1	(where international toll-free dial-up is supported)
	+1-708-848-7124	(other locations)
E-mail:	71303.3224@compuserve.com	(technical support only)

Visit out Web server at `http://www.lingo.com` for a live demo.

Last modified on February 10, 2000.

Tested on humans, not on animals.

IMPORTANT

Thank you for purchasing this Lingo Allegro USA, Inc. software product. It is important that you carefully read the following agreement prior to opening the disk envelope or using the software. Breaking the seal on the disk envelope or using the software indicates your agreement to be bound by the terms of this agreement. If you do not agree with them, do not break the seal or use the software and promptly return, unopened, the disk envelope and documentation for a full refund.

Lingo Allegro USA, Inc.

Software License Agreement

LICENSE GRANT. This is a license, not a sales agreement, between you and Lingo Allegro USA, Inc. ("Lingo"). Lingo grants to you a non-exclusive, non-transferable (except as provided below) license to use the copy of the software enclosed in the disk envelope and accompanying documentation in accord with the terms of this license agreement. The software is owned by Lingo and protected by copyright laws. Therefore, you must treat the software like any other copyrighted material except that:

You may:

a. Install the software on only one computer at a time. b. Make copies solely for backup purposes, provided that you include all proprietary notices on the copy. c. Transfer (but not rent or lease) the software on a permanent basis if the person receiving it agrees to the terms of this agreement and you do not keep any copies of the software or documentation for yourself.

You may not:

a. Use the software on more than one computer at a time. b. Modify, translate, reverse engineer, decompile, disassemble, create derivative works based on, or copy (except for backup purposes) the software or the documentation. c. Rent or lease any rights in the software or documentation in any form to any person without the prior written consent of Lingo which, if given, is subject to the transferee's consent to the terms and conditions of this license. d. Remove any proprietary notices, labels, or marks on the software, documentation, or containers. e. Transfer or disclose the software or documentation to anyone not bound by the terms of this agreement.

LIMITATION OF REMEDIES. Your sole remedy under this license agreement shall be repair or replacement as provided in the warranty. Lingo's sole and exclusive maximum liability for any claim by you or anyone claiming through or on behalf of you or arising out of your order or the warranty shall not in any event exceed the actual amount paid by you to Lingo for the product.

IN NO EVENT SHALL LINGO BE LIABLE FOR ANY INDIRECT, INCIDENTAL, COLLATERAL, EXEMPLARY, CONSEQUENTIAL OR SPECIAL DAMAGES OR LOSSES ARISING OUT OF YOUR ORDER OR DISKS DELIVERED UNDER IT OR OUT OF THE WARRANTY, INCLUDING WITHOUT LIMITATION LOSS OF USE, PROFITS, GOODWILL OR SAVINGS, OR LOSS OF DATA, DATA FILES, OR PROGRAMS THAT MAY HAVE BEEN STORED BY THE USER. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

LIMITED WARRANTY

THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE SOFTWARE IS WITH YOU, AND YOU ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. SOME STATES DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS, SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

Lingo does NOT warrant that the functions contained in the software will meet your requirements or that the operation of the software will be uninterrupted or error free. However, Lingo warrants the media disk on which the software is furnished to be free from defects in material and workmanship for a period of one (1) year from the date of shipment by Lingo to you. Lingo shall, at its option and cost, either repair or replace the media disk with a new or reconditioned disk provided the disk is returned by you to Lingo within the above warranty period.

30 DAY TRIAL. This software is provided on a 30 day trial basis. You agree to pay for this software unless, within 30 days of invoice date, you return the software and documentation, destroy any copies, and do not use the software after that time.

GENERAL. You agree to pay any taxes resulting from this agreement. This agreement shall be governed by the laws of the State of Illinois.

WARNING

Use of this software for providing LZW capability for any purpose other than using it as a part of Internet Server Auxiliary Processor is not authorized. The dynamic GIF images creation in the applications working under Internet Server Auxiliary Processor is not authorized unless user first enters into a license agreement with Unisys under U.S. Patent No. 4,558,302 and foreign counterparts. For information concerning licensing, please contact:

Unisys Corporation
Welch Licensing Department - C1SW19
Township Line & Union Meeting Roads
P. O. Box 500
Blue Bell, PA 19424, USA
LZW_INFO@UNISYS.COM

The LZW capability referred to in this warning is contained in the files isap.dll and isapg.dll and this patent licensing requirement is applicable only if you have purchased the dynamic GIF images creation option.

Table of Contents

1. Product Content and Installation

This product is meant to be used with Dyalog APL for Windows version 8.2.2 or later by Dyadic Systems Ltd. under the Microsoft® Windows NT™ Server 4.0 operating system. It works with Microsoft® Internet Information Server version 3.0 or later. Intel® Pentium™ processor is required. Internet Server Auxiliary Processor for Dyalog APL/W (ISAP) provides a high-level interface to Microsoft Internet Information Server (IIS). ISAP is a tool for writing application extensions for IIS using Dyalog APL/W.

Warning! ISAP requires Dyalog APL version 8.2.2 with the build date November 2, 1999 or later. The software will not work properly, if earlier version of the interpreter is used.

To install the product, stop ***all three*** Internet services: Gopher, FTP, and WWW using the "Services" applet in the Control Panel. Run the setup.exe program from distribution disk 1. The installation program creates the following directory structure:

```
<ISAP>                          - ISAP installation root directory (can be changed during setup)
        <ISERV>                 - application directory (APL part)
        <LINGO>                 - sample root Web directory (IIS part)
                <DOCS>          - HTML directory: /isapdoc virtual directory
                        <IMAGES>  - images directory
                <ISAP>          - executable directory: /isap virtual directory
isap.doc                        - this document
```

Key files installed by the setup program are:

- Dynamic link library named isap.dll, which is the ISAP executable module. This file should be copied into the directory on the Windows NT server where Internet users have "EXECUTE" access. We will call this directory the *ISAP home directory*. This directory has the /isap virtual name (alias).

- Dynamic link library olch3d32.dll, which is the server-side graphics support module. This file must be copied into the same directory where isap.dll file was copied.

- Text file isap.ini that is used for ISAP and APL configuration. This file must be in the ISAP home directory. It contains various settings for ISAP and Dyalog APL. See section *ISAP.INI File Settings* for details.

- Dyalog APL run-time files are copied into the ISAP home directory.

- HTML and image files that are used in the ISAP Demo are copied into the Lingo/Docs directory. The installation program registers this directory with IIS as the /isapdoc virtual directory. This directory should have both "READ" and "EXECUTE" access.

- The ISERV directory contains all APL-related files. This directory is invisible for Internet users. The Dyalog APL workspace file named iserv.dws contains the ISAP Application Manager, ISAP system functions, and a few sample ISAP applications. You might want to create your own APL workspace to manage Internet applications. ***In any case, there must be a startup APL workspace that will be loaded when ISAP is called by IIS for the first time.*** The text file iserv.ini is the configuration file for the Internet application server iserv.dws. Even if you are planning to write your own workspace that will manage APL applications, it is recommended to use such a file for storing global initialization information. It will significantly simplify the maintenance of the server. See section **ISERV.INI File Settings** for details.

- The dynamic link library named isapg.dll, which is an APL-side module of ISAP. This file ***must be*** in the same location where the application workspace (iserv.dws) resides.

- The text of this document in Microsoft Word 97 format named isap.doc.

Setup procedure:

1. Stop FTP, WWW, and Gopher (if installed) Internet services using the Control Panel/Services applet. If you are running IIS 4.0, make sure that IIS Admin Service is running.
2. Run setup.exe from the installation disk 1. Follow instructions on the screen.
3. Start WWW service using Control Panel/Services applet. Start Internet Service Manager (Manager Console) and verify that WWW service configured as follows. We assume that ISAP has been installed in the c:\isap directory):

- Physical directory c:\isap\Lingo\Docs uses IIS alias /isapdoc and has both "READ" and "EXECUTE" access.
- Physical directory c:\isap\Lingo\isap uses IIS alias /isap and has "EXECUTE" access.
- Files with the extension ".xml" are registered as script files for the script interpreter c:\isap\Lingo\isap\isap.dll.

The installation program creates a new program group "Internet Server Auxiliary Processor" (the group name can be changed during setup). The following items will be placed in this group:

- *ISAP Demo* – starts demo Web site.
- *Register ISAP With IIS* – runs registration script. This item is added, when the setup program detects IIS 4.0 or later. If you get an error during the setup procedure, you can try to run the registration process manually using this shortcut.
- *Uninstall ISAP* – removes ISAP from the computer. If you use IIS 4.0 or later, you have to run *Unregister ISAP with IIS* first.
- *UnregisterISAP with IIS* – runs unregistration script that removes ISAP from IIS. This item is added, when the setup program detects IIS 4.0 or later. You have to run this script before uninstalling ISAP.

If you want to run database demos, you must have the Microsoft *Northwind* demo database along with the Microsoft Access driver installed on your computer. Microsoft Office 97 installs these components by default. Before running the ISAP demo, please create **System DSN** named "MS Access" using the "ODBC32" applet in the Control Panel. This data source must use the MS Access driver. Select NORTHWIND.MDB as a database file. If you change the name of the data source, you must manually correct the DSN entry in the ISERV.INI file.

ISAP can run in one of two modes: production and development. In the production mode ISAP and Dyalog APL run, as a Windows NT service to reach the maximum performance. No user interaction is allowed in that mode. Both ISAP and Dyalog APL will be invisible to a user. If you want to use the NT server for development, you have to install the full version of Dyalog APL/W. If Dyalog APL was already installed before running the ISAP setup program, you should enter the path to its root installation directory when prompted by the installer. You can install Dyalog APL at later time and make changes to the isap.ini file manually. See section *ISAP.INI File Settings* for details. In development mode, the development version of Dyalog APL is run. The programmer can use anything that is available in the APL environment to create and to debug applications.

When you run ISAP Demo for the first time, ISAP works in production mode using the Dyalog APL run-time module that is installed from ISAP's distribution disk. If you have the full version of Dyalog APL installed on your computer you can use the Setup page available from the ISAP Demo application to switch between the production and development modes.

Uninstallation procedure:

1. Stop FTP, WWW, and Gopher (if installed) Internet services using Control Panel/Services applet. If you are running IIS 4.0 or later, make sure that IIS Admin Service is running.
2. Execute *Unregister ISAP with IIS* item from the Internet Server Auxiliary Processor program group. Skip this step for IIS 3.0 or earlier.
3. Execute Uninstall ISAP from the Internet Server Auxiliary Processor program group.

2. ISAP Programming Overview

Microsoft Internet Information Server (IIS) is a Windows NT Server service that provides three Internet services: FTP, Gopher, and WWW. ISAP is an interface between WWW Publishing Service and Dyalog APL.

2.1. CGI vs. ISAPI

Normally, the WWW service provides the transfer of static documents (usually in HTML format) using HTTP protocol. However, in some cases, when documents must contain dynamic information, static documents cannot be a satisfactory solution. In such cases, the documents must be created dynamically by programs that are called at the moment when a client requests them. A common way of doing this is so-called *Common Gateway Interface* (CGI). CGI is a set of rules that describe how an application should use standard input and standard output streams for processing HTTP requests. When a server receives a request that requires the execution of a CGI program, it doesn't send any document to the client. Instead, it starts the appropriate program that will create the required document dynamically. The program uses stdin/stdout to read the request's information and to write the final document to the client.

The main disadvantage of CGI is that every request requires a new copy of the CGI program to be executed in a separate *execution thread.* CGI programs are loaded and unloaded every time a new request needs to be processed. The *stdin/stdout* approach is not very effective either. For languages such as APL, which have very large run-time systems, this approach would be very slow and expensive.

Microsoft Internet Information Server provides a more effective alternative to CGI. Instead of executing .EXE modules, IIS allows the writing of *extensions* that use the so-called Internet Server Application Program Interface (ISAPI). IIS extensions are Windows DLLs that are loaded into IIS process space and become a part of the server. IIS loads extensions only once, when they called for the first time. They remain in memory until IIS is stopped, or it needs to free some space to run additional applications. Thus, extensions don't need to spend additional time for their initialization, as do CGI programs. IIS extensions use the server's functionality to perform data exchange with the client. They don't use the *stdin/stdout* method. As a result, IIS extensions work much faster than conventional CGI programs. For APL systems, the most important advantage is that the run-time system doesn't need to be reloaded every time to process a new request. A single copy of the APL run-time system can execute many different applications simultaneously.

Internet Server Auxiliary Processor (see Fig. 1) provides an interface between IIS and Dyalog APL that allows you to write IIS extensions in APL. When IIS receives an HTTP request that refers to ISAP.DLL, it loads the DLL into its process space, if it is not already loaded. For example:

http://myhost@mynet.com/cgi-bin/isap.dll?ws=my_app

ISAP.DLL also will be called when a client refers to any file with the extension that is registered with IIS as a *script file* for ISAP.DLL. By default, the installation program uses ".xml". For example:

http://myhost@mynet.com/doc/mydoc.xml

You can register a different file extension, if you want. More than one file type can be registered with IIS to be processed by ISAP.DLL. You should avoid using direct DLL calls in your HTML files. Even, if you need to produce not a text output (like image), you always should use a script file in order to invoke an APL program.

When ISAP.DLL is loaded into the IIS process space, the auxiliary processor starts Dyalog APL, using specifications in the ISAP.INI file. APL loads the application workspace that will handle the communications with ISAP. This workspace first must establish a connection with ISAP using the `ISAPStart` function. Next, the APL application performs its initialization and attaches functions from external DLLs, if needed. Finally, the application creates an invisible window and sends the **NOTIFY** command to ISAP. The window will receive notification messages from ISAP every time IIS is processing a request that refers to ISAP.DLL directly or indirectly. An APL application has at most 30 seconds to complete initialization. After initialization is complete, the application should process incoming notification messages from ISAP until a special termination message is received.

The *SHARE* function that can be found in the supplied sample workspace provides all necessary functionality to start an APL application. Please don't change anything in this function, unless you understand exactly what you are doing. You only need to place your custom initialization code into the *SETGLOBALS* function.

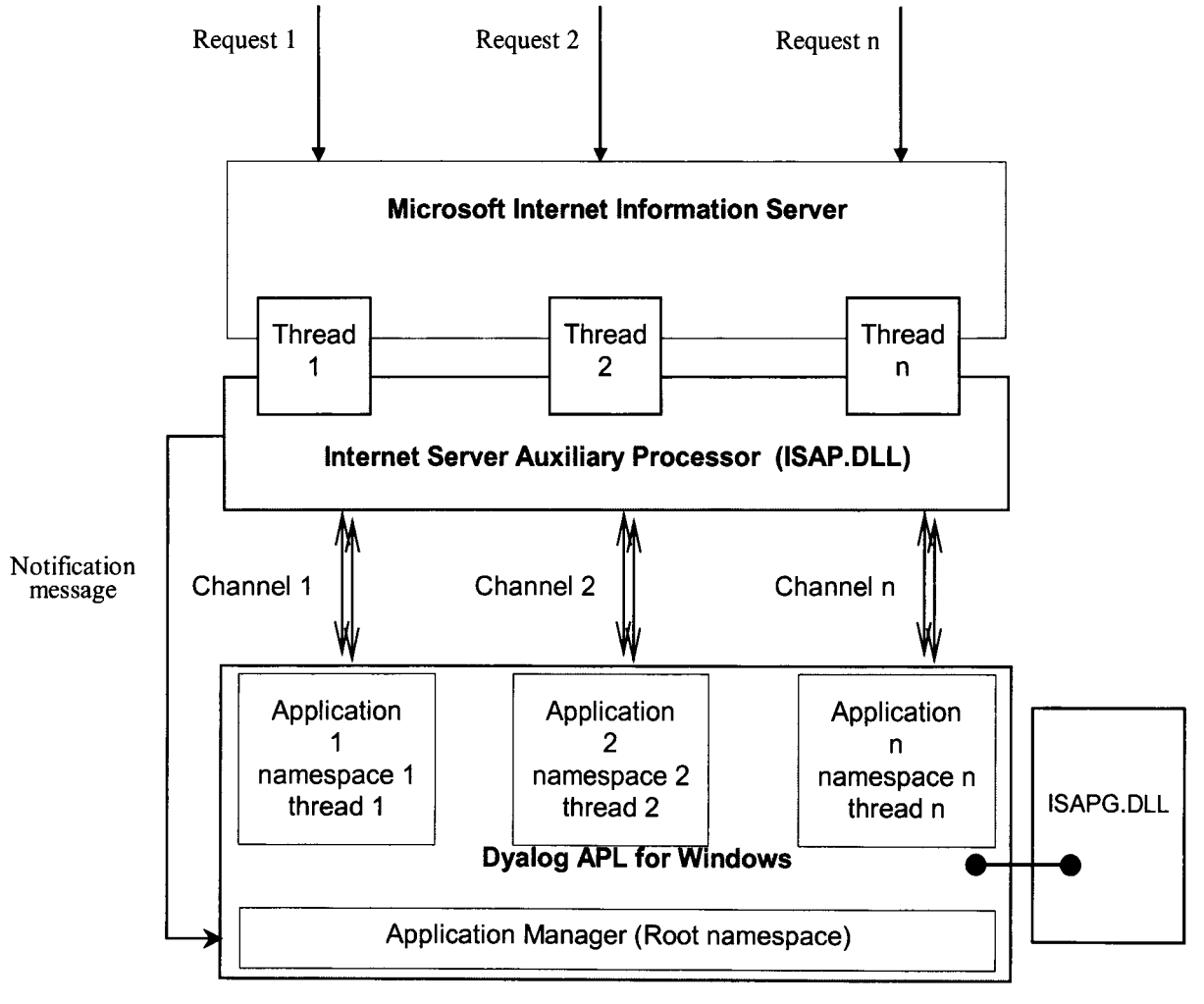

Figure 1. IIS - ISAP - Dyalog APL Interface

The notification window processes two messages: 95 and 131 (Windows shutdown). Message 95 is the notification message that ISAP sends to this window when a new *communication channel* is created (see below).

When IIS receives an HTTP request that requires ISAP execution, it starts a new execution thread and passes control to ISAP. If ISAP was called as the result of requesting a script file, ISAP loads the file into the *channel's buffer* and performs default processing of the script elements in this file, if there are any (see below). After that, ISAP creates a *communication channel* and sends a *notification message* to the APL application. The corresponding IIS thread is switching to commands processing mode. By receiving the notification message, the application can send *commands* to the channel to perform necessary tasks for processing the HTTP request. When the application closes the channel, ISAP terminates the corresponding IIS treads and completes the HTTP request.

ISAP's notification message is Dyalog APL event 95. Note that this message was introduced in Dyalog APL version 8.1.4. The message contains the channel number that has been created for the HTTP request. The APL application should be prepared to process many requests simultaneously. The notification message has the following structure:

MSG[1] - name of the notification window.
MSG[2] - Event code: 95.
MSG[3] - must be always 0.
MSG[4] - request's channel number; if it is 32767, the application must terminate.

The fourth item of the message is a *channel number*. Other items are not used. See the description of the **NOTIFY** command for details. The following *ISERV* defined function shows the minimal application startup code required:

```
      ISERV
[1] ⍝ INTERNAL INITIALIZATION
[2]   SETGLOBALS
[3] ⍝ CONNECT TO ISAP
[4]   →SHARE↓EXIT
[5] ⍝ START APPLICATION
[6]   ⎕DQ '.'
[7] ⍝ CLEAN UP AND DISCONNECTION FROM ISAP
[8]  EXIT:END
[9] ⍝ EXIT APL
[10]  ⎕OFF
```

The application itself is implemented as a callback function that is called every time ISAP's notification message is received.

Normally, an APL programmer does not need to know all the details above. The ISAP Application Manager handles everything described in this chapter. APL programmer works with ISAP on the level of programming of ISAP script files.

2.2. Channels, commands, functions, and template (script) files

Channel is the main concept in ISAP programming. Channel identifies the client who sent the HTTP request to ISAP. On the IIS side, each channel is supported by its own independently executed thread.

Each channel has an internal buffer on the server's side that is used to store a *template (script) file*. If ISAP was called as the result of referring to a script file (default extension ".xml"), this file will be loaded by ISAP in the channel's buffer *before* the APL application receives the notification message. Before sending the notification message to APL, ISAP performs default processing of the script file (see below). The file buffer also can be loaded explicitly by the application with any other file by using the **GETFILE** command. ISAP provides commands that allow manipulating this file without reading it into the APL workspace. The **TAGS** command returns all specified tags from the template file. The **REPLACE** command replaces a specified tag with text. The **FREPLACE** command replaces the specified tag with the contents of a file. The **PARTITION** command sends a fragment of the file to the client. The **HTML** command sends the entire processed template file to the client.

An APL program uses *commands* to request information from IIS regarding the HTTP request, to read data from the client, to load a template file, to process the template file, to create images, and to send the final document to the client. The *SAY* defined function, which can be found in the sample workspace, should be used to send commands to ISAP. This function takes an ISAP command as its argument and returns the result of the command as its explicit result. It generates error 667 when the command cannot be executed. APL applications must trap this error. The APL program must supply the channel number as one of the command's arguments, to allow data exchange with the client. (Some service commands do not require the channel number.)

The channel exists until the application explicitly closes it using the **CLOSE** command. Some ISAP commands close the channel implicitly. However, the application must execute the **CLOSE** command anyway. If the ISAP Application Manager is used, it will handle channel closing. When the channel is closed, ISAP informs IIS that processing of the request is complete. IIS terminates the corresponding execution thread and closes the TCP connection with the client.

In most cases an Internet application needs to respond with a dynamically created HTML file. The application could create such a file completely by itself. However, a much better approach is to use *template files*. Template files are HTML files created with standard Web publishing tools. They contain all necessary static HTML elements and can

be very complex. They also contain special *tags* that indicate where the dynamic content should be placed. Thus, the application needs to replace these tags in order to create the final document. The next section describes standard tags that ISAP processes automatically before the notification message is sent to the APL application. Users may also use custom, application-specific tags. The recommended tag format is: <!--#MYTAG-->, where MYTAG is any custom text.

Normally, APL programmer does not use most of ISAP commands directly. Instead, programmers should use ISAP *tags* and *functions*.

ISAP's *functions* are APL functions that can be executed in the application workspace. Functions supported by ISAPG.DLL are attached to the APL interpreter at start-up time. In most cases, ISAP's functions are wrapped by APL defined functions. Absolutely no modification is allowed to these cover functions. Lingo Allegro cannot guarantee that exported functions will be the same in later releases of ISAP. However, we guarantee that the syntax of the cover functions will remain the same. ISAP's functions extend the APL interpreter's functionality and allow a user to perform operations that could not be implemented purely in the APL environment, or would be too slow, if programmed in APL. These functions can be used anywhere in the applications.

2.3. How ISAP processes script files

By default, the installation program registers files with the extension ".xml" as ISAP's script files. As a result, when a client requests a file of this type from the HTTP server, IIS calls ISAP.DLL as a script interpreter. You can register different or more than one file type to be processed by the same (or another) instance of ISAP.DLL.

ISAP will load the script file into the channel buffer and will process the standard tags in this file before a notification message is sent to the APL application. For example, consider that the following file located in the /isapdoc virtual directory:

sample.xml:

```
<HTML>
<HEAD>
<TITLE>Internet Server Auxiliary Processor</TITLE>
</HEAD>
<BODY>
<CENTER>
<H2>Tags Example</H2>
<P>
<!--#INCLUDE /isapdoc/header.txt-->
<P>
Your IP address is: <!--#ISAP EXEC="_REMOTE_IP"-->
<!--#INCLUDE /isapdoc/footer.txt-->
<!--#ISAP FINISH=""-->
</BODY>
</HTML>
```

This HTML file contains four standard tags (shown in bold). When IIS receives the request:

GET /isapdoc/sample.xml HTTP/1.0

ISAP.DLL is called. ISAP will first replace the **INCLUDE** tags with the content of the appropriate files. Next it will generate an APL program to process the **EXEC** and **FINISH** tags. This program will be passed to the application manager for execution. If the header.txt and footer.txt are as follows:

header.txt:

```
<!-- This is /isapdoc/header.txt file -->
<P>
<FONT COLOR="#FF0000">Included header text.</FONT>
```

footer.txt:

```
<!-- This is /isapdoc/footer.txt file -->
<P>
<FONT COLOR="#0000FF">Included footer text.</FONT>
```

Then the client will receive the HTML file shown below. Note that in this case no APL programming was needed.

```
<HTML>
<HEAD>
<TITLE>Internet Server Auxiliary Processor</TITLE>
</HEAD>
<BODY>
<CENTER>
<H2>Tags Example</H2>
<P>
<!-- This is /isapdoc/header.txt file -->
<P>
<FONT COLOR="#FF0000">Included header text.</FONT>
<P>
Your IP address is: 209.19.63.2
<!-- This is /isapdoc/footer.txt file -->
<P>
<FONT COLOR="#0000FF">Included footer text.</FONT>

</BODY>
</HTML>
```

In most cases, an application will want to perform some application-specific processing of the script file. Developers should use standard **EXEC** tags or their own tags for these purposes. If a script file does not contain elements that require the execution of an APL program, ISAP does not call APL, but simply returns the file to the client in the form of a properly formed HTTP response. For example, if a file contains only **INCLUDE** tags, the APL interpreter is not called.

2.4. Processing the notification message

Usually, the APL Internet application needs to process requests of different types. It is practical to keep all such sub-applications as different namespaces. The central part of the APL application, called *the application manager*, will switch between those namespaces to support different requests from clients.

Individual APL applications (namespaces) can process HTTP requests in any order they wish, not necessarily in the order they have been received. The application manager should execute short requests first. In the simplest case, the APL application can process HTTP requests on a *"first come, first served"* basis and doesn't care about other, possibly pending, requests in its queue.

When the callback function (attached to the notification window) is called, it obtains the channel number from the notification message. Next, it uses the **GETINFO** command to receive information about the HTTP request. The result of **GETINFO** is a nested vector that contains everything the application possibly may need to process the request. If the HTTP `POST` method has been used, the application also receives the first 48Kb of the posted data from the client. For 99% of applications the **GETINFO** command is the only command that needs to be executed to obtain the client's data and the server's parameters. If the client sends more than 48Kb of data, the server application may call the **READ** command to get the rest of them.

If the client expects an HTML document, the application loads a template HTML file, if it is not loaded implicitly by ISAP, and replaces custom tags in the template file with dynamic content. If the client expects an image or a binary file, the application creates the corresponding output. Then the application sends the result to the client and closes the channel.

In most cases, when using an appropriate command, ISAP will send the result file with all necessary HTTP headers. However, the application can create the status line and all necessary HTTP response headers by itself using

HEADER and **WRITE** commands. The **HTML** command that sends the template file to the client, also allows to specify additional HTTP headers.

APL applications can use a Dyalog APL metafile object to create any required images. ISAP commands, **IMAGE**, **PIMAGE**, and **IMAGEX**, convert metafiles into GIF, PNG, and x-xbitmap formats. Those commands are implemented as defined functions that call ISAPG.DLL. ISAP also allows more effective operations with images using Windows APIs and functions from ISAPG.DLL. Those functions are attached to the application workspace by the *QNAStart* function.

If the output information is a short text document, the application can use the **HEADER** command to send the document along with custom headers. However, the general method is to use the **WRITE** command. It accepts any type of the output data and performs asynchronous data exchange with the client. The application doesn't have to wait until the data exchange is complete. For HTML files created from template files loaded in the channel buffer, use the **HTML** command to send the result document to the client.

When running in production mode, the APL application cannot interact with a user on the server by displaying windows and dialog boxes. Thus, it always should maintain its own log file to record all errors and other debugging information for future analysis. The application can use the **LOGWRITE** command to record short messages (up to 80 characters) directly into the Internet Server's log file.

The last action that the application should perform is to **CLOSE** the channel. The application supplies the status code that indicates how the processing of the request was completed. Correctly written applications should not use anything except "SUCCESS" and "SUCCESS AND KEEP CONNECTION OPEN". In case of error, the application should be able to send back to the client a properly formatted HTML that describes the reason why the request could not be processed. The APL application may use the **REDIRECT** or **SENDURL** commands for this purpose.

When the notification message specifies channel number 32767, the execution of the callback function and all applications ***must*** be terminated. This happens when the IIS is about to be stopped, or when it needs to free some system memory to run other requests. Normally this will not happen, except when the WWW service is shutting down. The application also should process message 131, which indicates Windows shutdown, in order to perform a graceful termination. See the supplied workspace for details.

Please note that the APL programmer is fully responsible for the quality of APL applications that are run under the IIS-ISAP interface. Any non-trapped APL error will stop all HTTP requests that refer to ISAP.DLL from being executed. In that case, it will be necessary to stop all three Internet services and manually kill the Dyalog APL process (sometimes it may require the reboot of the server) to make your application(s) functional again.

2.5. How the ISAP application manager works

The ISAP application manager is an APL workspace supplied with the product. The name of the workspace is *ISERV*. This workspace contains the application manager itself and sample namespaces (ISAP applications). The application manager handles all standard processing of HTTP requests, creates and deletes application threads, compiles programs for standard tags processing, and organizes simultaneous execution of the application threads. The application manager is a callback function that is executed by the APL interpreter when ISAP notification messages arrive.

When a new request that requires APL processing arrives to IIS, the notification message 95 is posted to APL. The callback function attached to this message starts a new APL thread by executing the *ISAPTHREAD* function, which implements the application manager, and exits.

First, the application manager executes the **GETINFO** command to obtain the request's parameters and data. Next, it creates a new namespace, where the *application thread* will be executed. If the request includes the name of the namespace that contains the APL code to be used for this request, this namespace is copied to the request namespace. Next, the application manager copies the content of the *ISAPCLI* namespace into client's namespace. This namespace contains standard functions, which must be present in each client's namespace. For example, functions that support standard tags. You can place your own common functions in this namespace. After that, the application manager defines APL variables to hold the request's environment variables. If the script file has **CVAR** and **NVAR** tags defined in it, the appropriate APL variables are created. Finally, the application manager compiles the

executable program using **EXEC** and **PART** tags found in the script file and the thread's latent expression, and executes the thread's program.

The thread's program is executed as a separated APL thread. This allows many application threads to be executed simultaneously. When the execution of thread's program completes, the application manager destroys the thread's namespace and closes the ISAP channel.

To add a new application, a programmer should follow the rules described later in this document. Users may develop their own application managers (not recommended). Lingo Allegro USA, Inc. will not support any user-created application managers other than the one supplied with the product.

3. ISAP Tags Reference

Tags are elements (strings) of an HTML file loaded in the channel buffer. When the HTML file that has the extension recognized by Internet Server as an ISAP script file (".xml" by default), is being loaded in the channel buffer, ISAP checks this file for the presence of standard tags. When the file is loaded in the channel buffer using the **GETFILE** command, no processing of the standard tags is performed. The *_PARSE* system function should be used to force the processing of explicitly loaded script files.

All standard tags begin with "<!--#ISAP" and end with "-->". An exception is the **INCLUDE** tag that begins with "<--#". ISAP processes standard tags before the APL application is called (the notification message is posted). Some tags are processed entirely by ISAP and do not require the execution of any APL code. For most of the tags, ISAP will generate an APL program that will be executed by the Application Manager. Some tags, like **CVAR**, **NVAR**, **PART**, and **FINISH**, are supported by the Application Manager itself. An APL programmer does not need to write the APL code for these tags. The **EXEC** tag refers an APL routine that should be written by the APL application programmer. Standard tags must appear in HTML documents exactly as it is shown in their descriptions. Any excessive embedded blanks cause an error.

The following standard tags are recognized and supported by ISAP:

CVAR defines the source and default value of a CGI character variable.

NVAR defines the source and default value of a CGI numeric variable.

COOKIE defines the source and default value of a HTTP cookie.

EXEC specifies an APL expression to be executed. Expression's result replaces the tag in the script file.

PART sends part of the document to the client.

FINISH instructs the application manager to send the output document to the client, when all tags are processed.

INCLUDE specifies the name of a file to be inserted in the result document instead of the tag.

WS defines the namespace to be used for the initialization of an application thread.

ISAP processes standard tags in the following sequence:

1. All **INCLUDE** tags, starting from the beginning of the script file.
2. The first **WS** tag found in the script file, starting from the beginning of the file.
3. All **CVAR**, **NVAR**, and **COOKIE** tags in order of their appearance, starting from the beginning of the script file.
4. All **EXEC** and **PART** tags in order of their appearance, starting from the beginning of the script file.
5. The first **FINISH** tag found in the script file, starting from the beginning of the script file.

ISAP executes standard tags in the following sequence:

1. The **WS** tag, if found, is used to initialize an application thread.
2. **CVAR**, **NVAR**, and **COOKIE** tags are used to initialize global variables in the application thread. If some variables were defined as the result of the thread's initialization, they will be re-defined. If one or more of these tags defines the same variable more than once, the last occurrence of the tag will be effective.
3. **EXEC** and **PART** tags are executed.
4. The thread's latent expression is executed (see chapter *Developing ISAP Applications* later in this document).
5. The **FINISH** tag is executed, if it was defined.

In the descriptions below, brackets "<" and ">" are used to denote meta elements defined in the tag's description. Brackets themselves are not part of the tag. All elements of the tags, excluding elements shown in brackets "<" and ">", must be in uppercase.

Programmers should avoid direct manipulations with the HTTP documents and use tags instead. Using tags make applications more transparent, easier to debug and to maintain. Good set of support APL functions allows to reduce the amount of routine APL programming to minimum and to replace it with the editing of script files.

3.1. COOKIE

Format: `<!--#ISAP COOKIE=<APL name> NAME="<Cookie name>" DEFAULT="<String>"-->`

Arguments:

`<APL name>`	properly formed name of an APL variable.
`<Cookie name>`	name of a cookie.
`<String>`	any character string without embedded single or double quotes.

Description:

This tag defines an APL variable with name `<APL name>` in the application thread. The value of the variable will be the value of the cookie `<Cookie name>`, if the client sent it. If the cookie was not defined within the HTTP request, the value of the APL variable will be set to the value of `<String>`.

Properly formed **COOKIE** tags are removed from the script file (replaced with blanks) before the APL application begins processing the file. **COOKIE** tags can be used anywhere in the script file. The application manager executes them before any **EXEC** tag found in the same script file. Thus, an **EXEC** tag can use, explicitly or implicitly, the value defined by a **COOKIE** tag, even if this **EXEC** tag is located before the **COOKIE** tag in the script file.

Example:

The following script file defines a simple CGI form that contains a single submit button. When a user clicks the button, the same script file is called. The application thread will be created using the *SAMPLE* namespace. The HTML document will display how many times user click the "Click Me" button on this page.

```
<HTML><BODY>
<!--#ISAP WS=SAMPLE-->
<!--#ISAP COOKIE=VISIT NAME="VIS" DEFAULT="0"-->
<CENTER>
You clicked the button <!--#ISAP EXEC="VISIT"--> times.
<BR>
<FORM METHOD=POST ACTION="/isapdoc/file.xml">
<INPUT TYPE=SUBMIT VALUE="Click Me">
</FORM>
</BODY></HTML>
<!--#ISAP FINISH="'VIS' #.ISAP.SETCOOKIE INC VISIT"-->
```

When a user clicks button first time, the cookie `VIS` is not defined yet. Thus, the thread variable VISIT will be assigned to the character vector `'0'`. The finish tag will execute the **#.ISAP.SETCOOKIE** function to set the value of the cookie and will send the appropriate HTTP header to the client. The *INC* defined function increments the value of the *VISIT* variable. This function should exist in the *SAMPLE* namespace and looks like follows:

```
     ∇ R←INC R
[1]    ⍎R←1+2⊃⎕VFI R
     ∇
```

3.2. CVAR

Format: `<!--#ISAP CVAR=<APL name> NAME="<CGI variable name>" DEFAULT="<String>"-->`

Arguments:

`<APL name>`	properly formed name of an APL variable.
`<CGI variable name>`	name of a CGI variable.
`<String>`	any character string without embedded single or double quotes.

Description:

This tag defines an APL variable with name `<APL name>` in the application thread. The value of the variable will be the value of the CGI variable `<CGI variable name>`. If the CGI variable was not defined within the HTTP request, the value of the APL variable will be set to the value of `<String>`.

CGI variable can be defined in the request's query string, or can be defined in the posted data. If the CGI variable was specified only once, the APL variable will have as its value a simple character vector. If the CGI variable was defined more than once, the value of the APL variable will be a nested vector of simple character vectors.

Properly formed **CVAR** tags are removed from the script file (replaced with blanks) before the APL application begins processing the file. **CVAR** tags can be used anywhere in the script file. The application manager executes them before any **EXEC** tag found in the same script file. Thus, an **EXEC** tag can use, explicitly or implicitly, the value defined by a **CVAR** tag, even if this **EXEC** tag is located before the **CVAR** tag in the script file.

Example:

The following script file defines a simple CGI form that contains an edit control and two submit buttons:

```
<HTML><BODY>
<!--#ISAP CVAR=TEXT NAME="txt" DEFAULT="No text"-->
<!--#ISAP CVAR=BUTTON NAME="BUTTON" DEFAULT="Button 1"-->
You typed <!--#ISAP EXEC="TEXT"-->.
<BR>
You clicked button <!--#ISAP EXEC="BUTTON"-->.
<BR>
<FORM METHOD=POST ACTION="/isapdoc/file.xml">
Type something and click a button:
<BR>
<INPUT TYPE=TEXT NAME="txt" VALUE="">
<BR>
<INPUT TYPE=SUBMIT NAME="BUTTON" VALUE="Button 1">  
<INPUT TYPE=SUBMIT NAME="BUTTON" VALUE="Button 2">
</FORM>
</BODY></HTML>
<!--#ISAP FINISH=""-->
```

When a user clicks one of the buttons, ISAP will start a new application tread using an empty namespace. Before any code is executed in the application thread, two global variables *TEXT* and *BUTTON* will be defined. The value of the first variable will be set to the value of the `txt` CGI variable (name of the edit control). The value of the second APL variable will set to the value of the `BUTTON` CGI variable i.e. "Button 1" or "Button 2".

When this file is loaded for the first time, that means the user hasn't clicked any button yet, and the APL variables will be initialized to their default values: *'No text'* and *'Button 1'*.

Two **EXEC** tags will display the values of *TEXT* and *BUTTON* variables.

3.3. EXEC

Format: `<!--#ISAP EXEC="<APL expression>"-->`

Arguments: `<APL expression>` an APL expression.

Description:

The **EXEC** tag is the main vehicle that drives ISAP programming. Strictly speaking, any type of server-side processing can be implemented by this tag. In degenerative case, the script file can contain only a single **EXEC** tag, which produce the entire output document. This tag defines an APL expression (usually, the name of a defined function) that will be executed by the application manager. The application manager will execute the string by surrounding it with quotes and applying the execute primitive: `⍎'expression'`. The APL expression may contain quotes to define APL character constants. For example, the following tag will produce a line of text `JUST A TEXT` in the HTML document:

```
<!--#ISAP EXEC="'JUST A TEXT'"-->
```

The APL expression must return a simple character vector or scalars `0` or `¯1`, as its explicit result. If any error occurs during the execution of the APL expression, an empty character vector will be used as the result of the expression. If the explicit result of the APL expression is not `0` or `¯1`, it replaces the corresponding **EXEC** tag in the script file.

If the APL expression returned *scalar* 0, the execution of the thread program (including the thread's latent expression) is terminated. The application manager will close the channel. If the application decided to interrupt the execution of the thread program (mainly because of an error), it should output some result document to the client using **HTML**, **WRITE**, **REDIRECT**, or other similar command.

If the APL expression returned *scalar* `¯1`, the current **EXEC** tag is ignored. Usually, applications use `¯1` when the script file is being switch dynamically under program control. See description of the **PARSE** command and **_PARSE** function later in this document.

EXEC tags can be used anywhere in the script file. If more than one **EXEC** tag is defined in the script file, they will be executed sequentially. The application manager executes **INCLUDE**, **WS**, **COOKIE**, **CVAR** and **NVAR** tags before any **EXEC** tag found in the same script file. Thus, an **EXEC** tag can use, explicitly or implicitly, values defined by **CVAR** and **NVAR** tags and global variables and functions defined in the namespace specified in the **WS** tag.

Any names used in `<APL expression>` refer to objects in the application thread, unless a proper namespace qualifier is used.

Example:

The following script file /isapdoc/file.xml defines a simple CGI form that contains an edit control and two submit buttons:

```
<HTML><BODY>
<!--#ISAP WS=SAMPLE-->
<!--#ISAP CVAR=TEXT NAME="txt" DEFAULT=""-->
<!--#ISAP CVAR=BUTTON NAME="BUTTON" DEFAULT="Button 1"-->
<FONT COLOR="#FF0000">
You typed <!--#ISAP EXEC="RHO TEXT"--> characters
and clicked <!--#ISAP EXEC="BUTTON"--> button.
</FONT>
<BR><CENTER>Type something and click a button:<BR>
<FORM METHOD=POST ACTION="/isapdoc/file.xml">
<INPUT TYPE=TEXT NAME="txt" VALUE="<!--#ISAP EXEC="TEXT"-->">
<BR>
<INPUT TYPE=SUBMIT NAME="BUTTON" VALUE="Button 1">  
<INPUT TYPE=SUBMIT NAME="BUTTON" VALUE="Button 2">
</FORM>
</BODY></HTML>
<!--#ISAP FINISH=""-->
```

When a user clicks one of buttons, ISAP will receive for processing the /isapdoc/file.xml script file defined by the HTML <FORM> tag. ISAP will start a new application thread using the *SAMPLE* namespace. Before any code is executed in the application thread, two global variables *TEXT* and *BUTTON* will be defined. The value of the first variable will be set to the value of the `txt` CGI variable (name of the edit control). The value of the second APL variable will set to the value of the `BUTTON` CGI variable i.e. "Button 1" or "Button 2". When this file is loaded for the first time, that means user hasn't clicked any button yet, so the APL variables will be initialized to their default values: `''` and `'Button 1'`.

Defined function *RHO* should be defined in the *SAMPLE* namespace:

```
      ∇ R←RHO TEXT
[1]     R←⍕⍴,TEXT
      ∇
```

This function returns the character representation of the length of its argument.

The tag `<!--#ISAP EXEC="RHO TEXT"-->` will output the length of the string that the user typed in the edit control before submitting the form. Note that the *RHO* function uses global the variable *TEXT* that has been defined by the **CVAR** tag. It is possible to refer to function RHO directly in its home namespace by using the namespace qualifier: `<!--#ISAP EXEC="#.SAMPLE.RHO TEXT"-->`.

The tag `<!--#ISAP EXEC="BUTTON"-->` simply displays the caption (value) of the button that the user clicked to submit the CGI form.

The tag `<!--#ISAP EXEC="TEXT"-->` is used to initialize the value of the edit control with the string that the user most recently submitted to the server.

The tag `<!--#ISAP FINISH=""-->` instructs the application manager to send the processed script file to the client without any additional HTTP headers (see the description of the **FINISH** tag).

The client's namespace always contains a few useful functions, which can be used inside the **EXEC** tag. Those functions allow performing standard routine procedures without any programming on the APL side. For example, if you need to restore selection of a radio button, made by a user, you could use the following code:

```
. . . . .
<!--#ISAP NVAR=fnc NAME="fnc" DEFAULT="0"-->
. . . . .
Select function to execute:<br>
<input type=radio name="fnc" value="0" <!--#ISAP EXEC="selected if fnc=0"-->>
- First<br>
<input type=radio name="fnc" value="1" <!--#ISAP EXEC="selected if fnc=1"-->>
- Second<br>
<input type=radio name="fnc" value="2" <!--#ISAP EXEC="selected if fnc=2"-->>
- Third<br>
. . . . .
```

Expression `"selected if fnc=0"` will return `selected`, if fnc is 0, and an empty string otherwise.

3.4. FINISH

Format: `<!--#ISAP FINISH="<APL expression>"-->`

Arguments: `<APL expression>` an APL expression.

Description:

This tag instructs the application manager that processing of the template (script) file is complete. The application manager will send the template file to the client using the **HTML** command. Any script file must contain one **FINISH** tag, if the result document needs to be send to the user. Script file may not contain the **FINISH** tag, if the document is sent implicitly by the thread latent expression (not recommended).

The `<APL expression>` (usually, the name of a defined function) specifies the APL expression that will be executed by the application manager to obtain additional HTTP headers. The application manager will execute the string by surrounding it with quotes and applying the execute primitive: `⍎'expression'`. The APL expression may contain single quotes to define APL character constants.

The APL expression must return a simple character vector as its explicit result. The CR-LF pair, including the last header must terminate returned headers. An empty vector returned by the APL expression is ignored. If any error occurs during the execution of the APL expression, an empty character vector will be used as the result of the expression. If HTTP headers have been sent already by previous **HEADER** or **PARTITION** commands, the result of the APL expression is ignored.

The tag can be located anywhere in the script file. Only the first **FINISH** tag found in the script file is processed by ISAP. All other **FINISH** tags, if they exist, are ignored. A properly formed **FINISH** tag is removed from the script file (replaced with blanks).

The application manager executes the **FINISH** tag after the execution of all **EXEC** tags completes and after the execution thread's latent expression. **EXEC** tags and application latent expression should not send the complete template file back to the user. However, they could send a part of the file using the **PARTITION** command or the **PART** tag. If the template file was sent and channel's buffer is empty, the **FINISH** tag will cause the error to be trapped by ISAP and the channel will be closed.

This tag allows the building of Internet applications from functions without using a latent expression (see Section *Developing ISAP Applications* later in this document).

Any names used in the `<APL expression>` refer to objects in the application thread, unless proper namespace qualifiers are used.

Example:

See example in the description of the **COOKIE** tag.

3.5. INCLUDE

Format: `<!--#INCLUDE <virtual path>-->`

Arguments: `<virtual path>` a virtual file name.

Description:

This tag instructs ISAP to include the specified file in the script file. The content of the included file replaces the tag in the script file. The `<virtual path>` parameter specified is the file name in IIS terms. For example, if the physical directory c:\InetPub\Docs is mapped to the virtual directory /doc on the HTTP server, the file c:\InetPub\Docs\MyText.txt must be referred to as /doc/MyText.txt.

The included file, in its turn, may contain one or more other **INCLUDE** tags. If the included file explicitly or implicitly refers to itself, there will be an infinite loop and IIS will have to be restarted. This is a duty of the programmer to avoid such situations. The included file may contain any other ISAP tags. They will be processed by ISAP when all **INCLUDE** tags have been processed.

The **INCLUDE** tag allows you to build output files from standard pieces. For example, if any page on a Web site must contain a standard copyright notice at the bottom, this element can be placed in a separate text file and included in the output files on the fly. If the notice should be changed, only the included file needs to be changed.

The **INCLUDE** tag is processed purely by ISAP. APL is not called if the script file does not contain any other tags.

Example:

The following script file uses two include files.

```
<HTML><BODY>
<!--#INCLUDE /doc/header.txt-->

. . . . . .

<!--#INCLUDE /doc/footer.txt-->
</BODY></HTML>
```

3.6. NVAR

Format: `<!--#ISAP NVAR=<APL name> NAME="<CGI variable name>" DEFAULT="<Number>"-->`

Arguments:

`<APL name>`	properly formed name of an APL variable.
`<CGI variable name>`	name of a CGI variable.
`<Number>`	character representation of a number or a numeric vector.

Description:

This tag defines an APL variable with name `<APL name>` in the application thread. The value of the variable will be the value of the CGI variable `<CGI variable name>`. If the CGI variable was not defined within the HTTP request, the value of the APL variable will be set to the value of `<Number>`.

CGI variables can be defined in the request's query string, or can be defined in the posted data. The application manager will try to convert the value of the CGI variable to a number. If it is not possible, the value 0 will be used. If the CGI variable was specified only once, the APL variable will have a value consisting of a one-element numeric vector. If the CGI variable was defined more than once, the value of the APL variable will be a simple numeric vector.

Properly formed **NVAR** tags are removed from the script file (replaced with blanks) before the APL application starts processing the file. **NVAR** tags can be used anywhere in the script file. The application manager executes them before any **EXEC** tag found in the same script file. Thus, an **EXEC** tag can use, explicitly or implicitly, the value defined by an **NVAR** tag, even if this **EXEC** tag is located before the **NVAR** tag in the script file.

Example:

The following script file /isapdoc/file.xml defines a simple CGI form that contains a select control and a submit button:

```
<HTML><BODY>
<!--#ISAP WS=SAMPLE-->
You selected <!--#ISAP EXEC="RHO Select"--> values.<BR>
<FORM METHOD=POST ACTION="/isapdoc/file.xml">
Select one or more values from the list and click the button:<BR>
<!--#ISAP NVAR=Select NAME="sel" DEFAULT="-1"-->
<select name="sel" multiple size="3">
      <option value="1">One
      <option value="2">Two
      <option value="3">Three
</select>
<BR>
<INPUT TYPE=SUBMIT NAME="BUTTON" VALUE="Click Me">
</FORM>
</BODY></HTML>
<!--#ISAP FINISH=""-->
```

When a user clicks one of the buttons, IIS will call ISAP using the /isapdoc/file.xml file as a script file. ISAP will start a new application tread using the *SAMPLE* namespace. Before any code is executed in the application thread, the global variable *Select* will be defined. The value of the variable will be set to the value of the `sel` CGI variable (name of the select control). If one option from the select is highlighted, the value of the variable will be a single corresponding number. If more than one option is selected, the value of the variable will be a vector of numbers. When this file is loaded for the first time, the user hasn't clicked the button yet, and the APL variable *Select* will be initialized to the default value `¯1`.

This example uses the **EXEC** tag with the *RHO* function. See the function definition in the description of the **EXEC** tag.

3.7. PART

Format: `<!--#ISAP PART="<APL expression>"-->`

Arguments: `<APL expression>` an APL expression.

Description:

This tag forces the output of a part of the document. The specified APL expression (usually, the name of a defined function) is executed. The application manager will execute the string by surrounding it with quotes and applying the execute primitive: `⍎'expression'`. The APL expression may contain quotes to define APL character constants.

The result of the expression is used as additional HTTP headers, which application wants to append to the standard HTTP headers. Every header line must be terminated with `CR` and `LF` characters (APL characters `⎕AV[4 3]`, also defined as a global variable *#.CRLF*). If the expression is omitted, or it returns an empty vector, no additional headers are sent. Note that application can create HTTP headers only once. HTTP headers are created explicitly by using the **HEADER** command. They are created implicitly by **HTML** command, or by previously executed **PART** tag. The result of APL expression is ignored, if HTTP headers have been already created. The additional headers, usually, specify HTTP cookies.

The application manager will sent a portion of the output document starting from the beginning of the file up to the beginning of the tag. The tag itself and the sent portion of the script file are removed from the channel's buffer. The **PART** tag allows outputting the resulting document in parts rather than as a single document. It can be used when some of the **EXEC** tags produce too long output. This may degrade the performance of the application, because the entire document is assembled on the server side. Application may instead to dump document in portions for static parts of the document and use **WRITE** command to write the output stream directly for large parts. See example below.

PART tags can be used anywhere in the script file. If more than one **PART** tag is defined in the script file, they will be executed sequentially. The application manager executes **INCLUDE**, **WS**, **COOKIE**, **CVAR** and **NVAR** tags before any **PART** or **EXEC** tags found in the same script file. Thus, an **PART** tag can use, explicitly or implicitly, values defined by **CVAR** and **NVAR** tags and global variables and functions defined in the namespace specified in the **WS** tag.

Any names used in `<APL expression>` refer to objects in the application thread, unless a proper namespace qualifier is used.

Example:

The following script file contains one **PART** tag and one **EXEC** tag:

```
<HTML><BODY>
<!--#ISAP WS=SAMPLE-->
Section 1 of the document.<br>
<!--#INCLUDE "/docs/rephead.txt"-->
<!--#ISAP PART=""-->

Section 2 of the document.<br>
<!--#ISAP EXEC="REPORT"-->

Section 3 of the document.<br>
<!--#INCLUDE "/docs/repfoot.txt"-->
</BODY></HTML>
<!--#ISAP FINISH=""-->
```

When the **PART** tag is executed, the first portion of the HTML file is sent. The *REPORT* function, used in the **EXEC** tag, produces a large report that is being sent during its generation using many **WRITE** commands. The *REPORT* function itself returns an empty vector. Thus, the output file is not growing. The **FINISH** tag will send the rest of the document (section 3).

3.8. WS

Format: `<!--#ISAP WS=<NS name>-->`

Arguments: `<NS name>` properly formed name of an APL namespace.

Description:

This tag defines an APL namespace that must exist in ISAP's application manager workspace. This namespace will be copied into the newly created application thread before any execution of the thread is started.

The namespace may or may not have a latent expression defined. The latent expression is a nested vector of simple character vectors. Each simple character vector must be a valid argument for the "execute" (⍎) primitive. The application manager will execute the latent expression after the execution of all **EXEC** tags is complete. See section *Developing ISAP Applications* later in this document for more details.

The **WS** tag can be located anywhere in the script file. Only the first **WS** tag found is processed. All others are ignored. A properly formed **WS** tag is removed from the script file (replaced with blanks) before the APL application starts processing the file.

If the **WS** tag was not defined in the script file, at least one **EXEC** tag must exist in the script file. If this is the case, the application manager creates an empty application thread that will contain only standard environment variables and, possibly, variables defined by **CVAR** and **NVAR** tags. If the **WS** tag is not defined and no **EXEC** tags are found in the script file, the script file is sent by ISAP to the client without calling APL (the ISAP notification message is not posted).

Example:

See examples in descriptions of **COOKIE**, **NVAR**, and **EXEC** tags.

4. ISAP Commands Reference

APL applications communicate with ISAP by sending *commands* using the *SAY* cover function. This function must be in the root namespace. No modifications are allowed to this function. All initialization that is needed in order to establish connection between APL and ISAP is performed by the *SHARE* defined function that must be located in the root namespace and must be called at the application's startup. The *SAY* function takes an ISAP command as its single argument, and returns the result of the command, if the command is successful. Error 667 is signaled to APL if an error occurs. The application ***must*** trap this error. Thus, the general syntax of ISAP commands, which we will use everywhere in this document, is:

```
DATA←SAY 'COMMAND' PAR1 PAR2 . . . PARn
```

The following commands are supported by the ISAP:

Null - null command. Doesn't have any functionality and is used for system purposes at startup.

CLEAR - removes all tags from loaded template file.

CLOSE - closes an HTTP channel and sends the request status to IIS.

CONVERT - converts data to the specified format.

FILESEND - starts fast asynchronous file transfer.

FREPLACE - replaces a pattern in the template file with the content of another file.

GETFILE - externally reads template file.

GETINFO - gets the IIS environment variables and client's data.

HEADER - sends HTTP status line and response headers to the client.

HTML - sends an HTML document, prepared from a template file.

LOGWRITE - records status and debugging information into the Internet Server's log file.

NOTIFY - specifies the window that should receive notification messages from ISAP.

MAPURL - maps URL path into physical path on the server's computer.

PARSE - loads and parses new script file.

PARTITION - sends a part of the template file to the client.

READ - reads data from the client, if **GETINFO** didn't supply all the client's data.

REDIRECT - instructs the client to use another URL (possibly, on a different server), instead of a given one.

REPLACE - replaces sub-strings in the template HTML document.

SENDURL - instructs IIS to send to the client an URL, located on the same NT server.

TAGS - returns specified tags from the template file.

TIME - returns properly formatted HTTP date/time information relative to the current time.

WRITE - sends data to the client.

3DBAR - asynchronously draws 3D bar chart.

3DCHART - asynchronously draws 3D surface chart.

The rest of this chapter gives detailed descriptions of all ISAP commands. Parameters in brackets ("[" and "]") are optional and can be omitted.

4.1. NULL operation

Format: *STATUS←SAY ''*

Arguments: None

Result: *STATUS* - is 0, if ISAP is ready.

Description:

This operation returns the current status of ISAP. It is used by the *SHARE* function only. ISAP returns 0 if it was properly initialized.

Example:

```
      □←SAY ''
0
```

4.2. CLEAR

Format: *res←SAY 'CLEAR' chan*

Arguments: *chan* - integer scalar; channel number;

Result: *res* - integer scalar; always zero;

Description:

This command removes all tags from the template file currently loaded in the channel buffer. The template file could be loaded by ISAP implicitly, if the HTTP request reefers ISAP's script file. Or it could be loaded explicitly by the **GETFILE** command. The CLEAR command removes all tags that have the following format:

```
<!--#any string-->
```

This command can be used when the application decides to abort the execution of the HTTP request, in order to remove all tags from the template file before sending it to the client.

Example:

```
SAY 'CLEAR' 0
```

4.3. CLOSE

Format: *res←SAY 'CLOSE' chan status*

Arguments:

chan	- integer scalar; channel number;	
status	- integer scalar, request termination status; this argument must be one of the following:	
	0 - "SUCCESS";	
	1 - "SUCCESS AND KEEP CONNECTION OPEN";	
	2 - "ERROR";	

Result: *res* - integer scalar; always zero;

Description:

This command shuts down the communication channel. It always must be the last command for a given HTTP request. When ISAP receives this command, it terminates the appropriate request thread and sends the specified status code to IIS. If you use status 1, you must send the correct `Content-Length` HTTP header with the output document. Don't use status 1 if you don't understand what you are doing. The **CLOSE** command can be used immediately after the **WRITE** command. ISAP guarantees that no data will be lost.

Applications running under the Application Manager should not use the **CLOSE** command directly. The application manager closes the channel automatically when thread program terminates.

Example:

```
SAY 'CLOSE' 5 0
```

4.4. CONVERT

Format: `res←SAY 'CONVERT' tt vect`

Arguments:
- `tt` - character scalar: specifies the type of conversion. The following values are allowed:
  - `'A'` - conversion from character APL vector to ANSI vector.
  - `'N'` - conversion from ANSI vector to APL character vector.
  - `'H'` - conversion from ANSI vector to parsed APL array.
- `vect` - a simple character vector.

Result: `res` - a simple character vector for `'A'` and `'N'` conversions, and a nested vector of simple character vectors for `'H'` conversion.

Description:

This command converts APL character vectors and scalars into their representation in ANSI code and back. It also parses and converts character vectors in ANSI code which have CGI-style query syntax into nested vector of simple character vectors in Dyalog APL code.

`'A'` and `'N'` conversions don't require long explanations. They simply convert character data between Dyalog APL and ANSI code tables. You will almost never need this feature, unless you do something very specific, or use some bad ideas. All ISAP commands that provide data exchange have their own conversion capabilities. If you are going to transfer binary information, don't use conversions from/to APL code and back. It may alter the data. For example, if *V* is a character vector in ANSI code (result of `⎕NREAD`, for example), the following expression may alter the value of *V*:

```
V←SAY 'CONVERT' 'A' (SAY 'CONVERT' 'N' V)
```

The `'H'` conversion should be used when you process CGI-formatted query strings and/or posted data from the client. Such lines use the "&" character as a token delimiter. Some special characters ("+", "/", "&", etc.) are used by the HTTP protocol itself. Thus, they cannot be used in request strings or in posted data. Browsers replace them with *escape sequences*. For example, if you want to send the request:

```
http://myserv@my.net/isap/isap.dll?WS=COUNTER&EXPR=2+2
```

as the result of using an HTML form, the browser will translate this request and the server receives the following query string:

```
WS=COUNTER&EXPR=2%02B2
```

The `'H'` conversion will parse the query string for you:

```
      QS←SAY 'CONVERT' 'H' QS
      ⍴QS
2
       QS
 WS=COUNTER  EXPR=2+2
```

So, the `'H'` conversion will split the original vector at each "&" character and will replace all escapes with the appropriate single characters. Again, as in the case of `'A'` and `'N'` conversions you will almost never need this, unless you have to use the **READ** command. For most applications, the only data input command that needs to be executed is the **GETINFO** command that provides its own translation capabilities.

Example:

```
      A←SAY 'GETINFO' CHAN 'N'
      ⎕←SAY 'CONVERT' 'H' ((2⊃A),19⊃A)
 CALC.HTM  WS=COUNTER  EXPR=2+2
```

4.5. FILESEND

Format: *res←SAY 'FILESEND' chan file [(tt head tail)]*

Arguments:

chan - integer scalar; channel number;
file - simple character vector; fully qualified file name.
tt - character scalar; translation mode. Allowed values are:
'*A*' - specifies translation from APL to ANSI.
'*N*' - no translation.
head - simple character vector; data to send before the file's content.
tail - simple character vector; data to send after the file's content.

Result: *res* - integer scalar; always 0.

Description:

This command starts a very fast and effective file transfer operation on channel *chan*. The *file* parameter specifies the file to be transferred. The application may use the **HEADER** command before calling **FILESEND**, if it wants to send any HTTP information before the file transfer (content type, for example). The APL application ***must*** **CLOSE** the channel after using the **FILESEND** command.

If the third parameter present, it must be three-item nested vector that specifies the translation mode and two character vectors. The *head* vector will be sent to the client before the content of the file. The *tail* vector will be sent after transmitting the content of the file. In both cases the *tt* translation mode will be used. Any one (or both) of these vectors can be empty.

This command can be used for effective file transfer operations under application program control.

Example:

```
SAY 'FILESEND' CHAN 'c:\myprog\myprog.exe'
SAY 'FILESEND' CHAN 'c:\test.txt'('A' '***START***' '***END***')
```

4.6. FREPLACE

Format: `res←SAY 'FREPLACE' chan pat file`

Arguments:
- `chan` - integer scalar: channel number.
- `pat` - character vector: pattern to replace.
- `file` - character vector: fully qualified file name.

Result: `res` - integer scalar: always 0.

Description:

The **FREPLACE** command replaces the pattern `pat` with the content of the file specified by the `file` argument in the template file that is currently loaded in the channel's buffer. Channel is specified by the `chan` argument. The template file could be loaded implicitly by ISAP, as a result of processing of the HTTP request that refers to the file with the extension "xml", or it could be loaded explicitly by the application using the **GETFILE** command on the same channel. All occurrences of the pattern `pat`, if any, will be replaced. The pattern vector cannot be empty.

The `file` argument must specify the fully qualified path to the file. If an application works with IIS virtual directories (recommended) the virtual path can be translated into the physical path using **MAPURL** command. The **FREPLACE** command is executed asynchronously in a separate execution thread. In all cases, when possible, the **INCLUDE** tag should be used instead of the **FREPLACE** command.

Example:

The following example implements the INCLUDE tag. The defined function *INCLUDE* below scans the currently loaded template file on a given channel *CHAN* and replaces all tags that have the following format:

```
<!--#INCLUDE FILENAME-->
```

Where *FILENAME* specifies the virtual path to the file whose content will replace the tag.

```
      FLAG←INCLUDE CHAN;FILE;TAGS;A;⎕TRAP
[1]   ⍝ GET ALL INCLUDE TAGS FROM THE FILE
[2]    →(0=⍴TAGS←SAY 'TAGS' CHAN '<!--#INCLUDE ' '-->')↑0
[3]   ⍝ MAP FILE NAMES INTO PHYSICAL NAMES
[4]    A←⊂¨SAY¨(⊂'MAPURL' CHAN),¨(⊂¨TAGS)
[5]   ⍝ CREATE ARGUMENT FOR FREPLACE
[6]    TAGS←⊂¨(⊂'<!--#INCLUDE '),¨TAGS,¨⊂'-->'
[7]    TAGS←∪TAGS,¨A
[8]   ⍝ REPLACE ALL TAGS
[9]    SAY¨(⊂'FREPLACE' CHAN),¨TAGS
```

Example of the template file:

```
<HTML>
<HEAD><TITLE>INCLUDE Tags Demo</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000" LINK="#0000FF" VLINK="#551A8B">
<FONT FACE="COMIC SANS MS, ARIAL">
<CENTER>
<!--#INCLUDE /doc/include.txt-->
<P>
Your IP address is: <FONT COLOR="#FF0000"><!--#IP--></FONT>.
</BODY></HTML>
```

When the above function is applied to this template file, the **INCLUDE** tag will be replaced with the content of the file with the virtual path `/doc/include.txt`. See also descriptions of the **TAGS** and **MAPURL** commands.

4.7. GETFILE

Format: *len←SAY 'GETFILE' chan [fname]*

Arguments: *chan* - integer scalar; channel number;
fname - character vector; fully qualified path to a file.

Result: *len* - integer scalar: length of the read file in characters, or character vector: the current content of the template file.

Description:

When the *fname* parameter is present, this command loads the file specified by the *fname* argument into the channel's buffer. The file can be altered using the **REPLACE, FREPLACE**, and **PARTITION** commands. The **TAGS** command can be used for analysis of that file. The result of the command is the length of the file. Each opened channel can have its own file associated with that channel. The application can use the **GETFILE** command more than once during processing of the same request. Each execution of **GETFILE** replaces the previously read file with a new one.

The **GETFILE** command should be used when the application cannot determine what template file to use before it starts processing the request. In such cases, the name of the template file is passed as a parameter of the request. When the template file is known in advance, it should use the extension registered with IIS as a script file for ISAP (by default, ".xml") and the HTTP request should refer to it explicitly rather than refer to the ISAP.DLL file. The use of an "xml" file is more effective because ISAP loads such files implicitly without processing on the APL side. For example, the following HTTP requests are equivalent:

```
http://www.lingo.com/isap/isap.dll?WS=MYPROG&FILE=MYFILE.XML
http://www.lingo.com/doc/myfile.xml?WS=MYPROG
```

The first request will require use of the **GETFILE** command to load the template (script) file. The second request does not require the execution of the **GETFILE** command, because ISAP will load it before posting the notification message. The standard tags (see chapter *ISAP Tags Reference*) are not processed by ISAP when the **GETFILE** command is used to load the script file. We assume that the name of the template file to use is passed as a value of the `FILE` parameter and the template file `MYFILE.XML` is located in the virtual `DOC` directory.

When the *fname* parameter is omitted, the **GETFILE** command returns the current content of the stored file associated with channel *chan*. The application should avoid reading the file into the workspace.

The APL application should not generate a complete HTML document by itself. Instead it should use template documents that contain all necessary formatting like colors, fonts, graphics, links, and so on. The template document includes application-specific tags that the application program should replace with dynamic content using **REPLACE** and **FREPLACE** commands. The application can use the **TAGS** command to analyze the template file. When all tags are replaced, the application executes the **HTML** command to send the result document to the client. Such an approach makes the application code independent from the actual form of the result document. Common HTML editors can be used to edit the template documents. It also increases the performance of the APL application, because it doesn't have to read and process the template file in the workspace. Applications can also use the **PARTITION** command to send a part of the template file to the client.

The recommended tag format is: `<!--#MYTAG-->`. Such a tag is not visible when the HTML document is displayed in the Web browser, if it is not replaced. This can be used for default processing. The "#" character helps to make the tag unique. The template document can contain any number of custom tags.

Example:

See examples in descriptions of **HTML** and **REPLACE** commands.

4.8. GETINFO

Format: `A←SAY 'GETINFO' chan [tt]`

Arguments:
`chan` - integer scalar: channel number.
`tt` - character scalar: specifies the type of conversion. The following values are allowed:
`'A'` - conversion from character APL vector to ANSI vector.
`'N'` - conversion from ANSI vector to APL character vector.
`'H'` - conversion from ANSI vector to parsed APL array.
This parameter can be omitted. If so, the `'H'` conversion is assumed.

Result:
`A` - nested 20-item vector:
`1⊃A` - simple character vector - *path translated.* Physical path to the requested document.
`2⊃A` - five-item nested vector. ISAP system info:
`1⊃2⊃A` integer scalar `1`, if the template (script) file was loaded by the ISAP in the channel's buffer. `0` - if the file was not loaded. Last case means that the ISAP.DLL has been referred to explicitly in the HTTP request.
`2⊃2⊃A` simple character vector. The value of the **WS** tag found in the script file. This will be an empty character vector, if the **WS** tag was not defined in the loaded script file, or the script file was not loaded (the previous element is `0`).
`3⊃2⊃A` nested vector of simple character vectors or an empty vector. Used by the application manager to compile the thread's program from **EXEC** tags found in the script file.
`4⊃2⊃A` nested vector of simple character vectors or an empty vector. Used by the application manager to set the thread's global variables from **CVAR** and **NVAR** tags found in the script file.
`5⊃2⊃A` nested vector of a simple character vector or an empty vector. Used by the application manager to support the **FINISH** tag, if it exists.
`3⊃A` - simple character vector: *request method.* Can be *GET* or *POST*.
`4⊃A` - simple character vector, if `'A'` or `'N'` translations are used, or nested vector of simple character vectors, if `'H'` translation is used - *query string.* See the description of the **CONVERT** command for the explanation of the translation modes.
`5⊃A` - simple character vector: *path info.*
`6⊃A` - simple character vector: *content type.* The type of data, if `POST` method used.
`7⊃A` - simple character vector: *remote address.*
`8⊃A` - simple character vector: *remote host.*
`9⊃A` - simple character vector: *referrer URL.*
`10⊃A` - simple character vector: *remote user*;
`11⊃A` - simple character vector: *unmapped remote user.*
`12⊃A` - simple character vector: *server name.*
`13⊃A` - simple integer scalar: *server port.*
`14⊃A` - simple integer scalar: *server port secure.* This will be `1`, if request submitted via secure connection, or `0` otherwise.
`15⊃A` - simple character vector: *base portion of original URL.*
`16⊃A` - simple character vector: *HTTP accepts*: values of the `Accept` header divided by ",".
`17⊃A` - simple character vector: *authentication type.*
`18⊃A` - nested matrix n by 2, each element of which is a simple character vector. The first column is *HTTP header names.* The second column is the corresponding *header values.* This element of the result returns all HTTP headers received with the request.
`19⊃A` - simple integer scalar: number of additional bytes pending on client's side. The number of bytes that can be read by the **READ** command.
`20⊃A` - simple character vector, if `'A'` or `'N'` translations are used, or nested vector of simple character vectors, if `'H'` translation is used. This is the data posted by the client, if the `POST` method is used. See the **CONVERT** command for the explanation of conversion modes. The size of this item is up to 48Kb. If the client posted more than

48Kb, it will be indicated by the 19th item of the result.

If some server variable is undefined, or it doesn't make sense for a given request, the corresponding item of *vars* will be an empty character vector. For all character items, unless it is explicitly indicated, the translation ANSI-to-APL is applied.

Description:

This is the first command that the APL application should execute before it can start the processing of the HTTP request. For most applications this is the only input command that is required. The **GETINFO** command should be used in application managers only. Application threads should be provided with the result on this command during their initialization. Calling the **GETINFO** command more than once significantly reduces the overall performance.

If **GETINFO** didn't return all the data sent by the client (item 19 of the result is not zero), the application can use the **READ** command to receive additional data. Most often-used server variables are kept in separate items of the result that makes the programming easier. If you need to get any additional variables (HTTP headers), you may parse item 18 of the result. It contains all HTTP headers received with the request.

If the ISAP.DLL is called by the IIS because the client requested a script file, ISAP will load that file in the channel's buffer before sending the notification message to the APL application. The first element of item 2 of the result will have value 1 in this case and other elements of this item will be set as the result of processing the script file. The application can explicitly load another template file, if necessary, using the **GETFILE** command.

Example:

```
      A←SAY'GETINFO' 0
      A[3 1 7 20]
POST  home.htm  146.240.91.85   WS=COUNTER  EXPR=5*4
```

4.9. HEADER

Format: `res←SAY 'HEADER' chan stat [hdrs]`

Arguments:
`chan` - integer scalar; channel number;
`stat` - simple character vector; valid HTTP status line, or empty vector;
`hdrs` - simple character vector; additional HTTP headers and, possibly, body of the document.

Result: `res` - always integer scalar 0.

Description:

This command sends standard headers to the client: server type, MIME version, and the creation date along with any custom headers specified. The `stat` parameter must represent a valid HTTP response status line. For example: `'200 OK'`. If this parameter is an empty vector, the response status line `'200 OK'` will be sent.

The second argument specifies additional headers for the response. Each header should be terminated with `⎕AV[4 3]`. The last header should be terminated with the sequence `⎕AV[4 3 4 3]`. The application may supply other text data after the last header. These data will be treated as a response body. This method is not recommended. Applications should use **WRITE**, **HTML**, or **PARTITION** commands to send the response body. If the second argument is omitted, IIS sends standard headers only.

Only one **HEADER** command can be used per request. If any other command that sends HTTP headers has been used during the processing of the request, the **HEADER** command cannot be executed.

Example:

The following defined function sends an HTML document supplied as its argument:

```
      CHAN SENDHTML DOC;HDRS
[1]   HDRS←'Content-Type: text/html',⎕AV[4 3]
[2]   HDRS,←'Content-Length: ',(⍕⍴DOC),⎕AV[4 3]
[3]   HDRS,←'Last-Modified: ',(SAY 'TIME'),⎕AV[4 3 4 3]
[4]   SAY 'HEADER' CHAN '' HDRS
[5]   SAY 'WRITE' CHAN 'A' DOC
```

4.10. HTML

Format: *res←SAY 'HTML' chan [hdrs]*

Arguments: *chan* - integer scalar: channel number.
hdrs - character vector: additional HTTP headers.

Result: *res* - integer scalar: always 0.

Description:

The **HTML** command sends to the client the current content of the channel's buffer, as an HTML document. This file could be implicitly loaded by ISAP, if the client's request refers to the file type registered as a script file (default is ".xml"), or it could be explicitly loaded by the application using the **GETFILE** command on the same channel. The **HTML** command sends all necessary standard HTTP headers and the status line.

If **HEADER** or **PARTITION** commands were executed on the same channel, no status line and HTTP response headers are sent to the client. The *hdrs* argument is ignored in this case.

The command sends the following standard HTTP headers: `Content-Type`, `Content-Length`, `Date`, `Expires`, `Last-Modified`, `Pragma: no-cache`. The expiration date is set in the past. The length of the document is calculated by ISAP. The application can specify additional headers in the *hdrs* parameter. Each header, including the last one, must be terminated with the `CR-LF` pair.

The application must close the channel using the **CLOSE** command after executing the **HTML** command.

If an application uses the **HTML** command inside a defined function that implements an **EXEC** tag, the function should return 0 as its explicit result. The application manager will terminate the current thread program and will close the channel. If an application uses the **HTML** command inside thread latent expression, the application should explicitly reset thread program by setting the *_COM* global variable to an empty vector.

The **HTML** command provides an alternate and more effective way of sending HTML documents from ISAP's internal buffer in comparison with the **HEADER-WRITE** sequence, which assumes that the result HTML document is generated in the application workspace.

Example:

The following is an example of the template HTML file c:\InetPub\doc\ipdemo.htm:

```
<HTML>
<HEAD><TITLE>Template HTML File Usage Demo</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000" LINK="#0000FF" VLINK="#551A8B">
<FONT FACE="COMIC SANS MS,ARIAL">
<CENTER>
<H1>Client IP Address Demo</H1>
<P>
Your IP address is: <FONT COLOR="#FF0000"><!--#IP--></FONT>.
</BODY></HTML>
```

The following APL code example reads the template file, obtains the client's IP address as a character vector as the result of the *GETIPADDR* defined function, replaces tag `<!--#IP-->` with the actual value and sends the result HTML document to the client:

```
SAY 'GETFILE' CHAN 'C:\INETPUB\DOC\IPDEMO.HTM'
SAY 'REPLACE' CHAN '<!--#IP-->' GETIPADDR
SAY 'HTML' CHAN ('Set-Cookie:visit=',(⍕VIS←VIS+1),⎕AV[4 3])
```

See also descriptions of **REPLACE** and **GETFILE** commands.

4.11. LOGWRITE

Format: `len←SAY 'LOGWRITE' chan msg`

Arguments: `chan` - integer scalar: channel number.
`msg` - character vector: message to write.

Result: `len` - integer scalar: number of bytes written into the server's log.

Description:

This command writes a diagnostic message into the Internet Server's log file. The message will appear as the element of the query string of the request that is being executed on the channel `chan`. The length of the message is limited to 80 characters. If the `msg` is longer than 80 character, it will be truncated

The use of the IIS log file to write tracing information is useful for maintenance purposes. However, the application always should create its own application log to record all errors and other debugging information using APL system functions that work with native files. This is the only way the production mode application can communicate with a developer.

Example:

Writing a message into IIS log file:

```
SAY 'LOGWRITE' CHAN 'This message will be placed into IIS log'
```

4.12. NOTIFY

Format: `res←SAY 'NOTIFY' handle`

Arguments: `handle` - integer scalar; the value of the HANDLE property of a "Form" object;

Result: `res` - integer scalar; always 0.

Description:

This command assigns a Dyalog APL window (form) to receive notification messages from the ISAP. The single argument of this command should be obtained as the result of

```
'myform' ⎕WG 'HANDLE'
```

Application must attach a callback function to event 95. ISAP will post the notification message to the specified window every time a new HTTP request is received from IIS. The format of the notification message `MSG` is as follows:

`MSG[1]`	- Object name.
`MSG[2]`	- Event code: 95.
`MSG[3]`	- must be always 0.
`MSG[4]`	- request's channel number; if it is 32767, the application must terminate.

The technique of using this command should be as described below:

1. Create an ***invisible*** window (form) using the `⎕WC` function.
2. Attach a callback function to event 95 for the created window. This function will process HTTP requests. It also should process user defined message 1001, that will be used for switching between applications, if you choose to use this method as described later in this document.
3. Issue a **NOTIFY** command. Wait for messages (execute `⎕DQ`).
4. Issue the appropriate ISAP commands in response to HTTP requests, as described in chapter 2.
5. Expunge the form, terminate all applications, and exit APL, when channel number 32767 is received.

The **NOTIFY** command should be used at startup time in application managers only.

Example:

See the `SHARE` defined function in the supplied workspace.

4.13. MAPURL

Format: *path←SAY 'MAPURL' chan vpath*

Arguments: *chan* - integer scalar: channel number.
vpath - character vector: virtual path to translate.

Result: *path* - character vector: physical path on the server's computer.

Description:

The **MAPURL** command translates a virtual path into a physical path. Parameter *chan* can be any valid channel number. This command allows writing programs that don't depend on physical location of files on the server's computer.

Example:

The results of execution of the following expressions will depend on the assignments of the virtual directories on the Internet Server:

```
      ⎕←SAY 'MAPURL' 0 '/'
C:\InetPub\Lingo
      ⎕←SAY 'MAPURL' 0 '/doc/default.htm'
C:\InetPub\Lingo\Doc\default.htm
      ⎕←SAY 'MAPURL' 0 '/images/lingo.gif'
C:\InetPub\Lingo\Doc\Images\lingo.gif
```

4.14. PARSE

Format: *ws vars exec fin←SAY 'PARSE' chan filename*

Arguments:

	chan	integer scalar: channel number.
	filename	character vector: fully qualified physical path to a script file.

Result:

ws	simple character vector or an empty vector: namespace name.
vars	nested vector of simple character vectors or an empty vector: APL program to define CGI variables specified by **NVAR** and **CVAR** tags.
exec	nested vector of simple character vectors or an empty vector: APL program to support **EXEC** tags defined in the script file.
fin	nested one-item vector that consists of a simple character vector or an empty vector: APL program to support the **FINISH** tag defined in the script file.

Description:

The **PARSE** command loads the script file specified by the *filename* parameter into the file buffer on the channel *chan* and performs the standard tags processing. All **INCLUDE** tags are replaced with the content of the appropriate files. The command returns the application manager system info, similar to the second element of the result of **GETINFO** command. If file *filename* could not be found, an error is generated and the content of the channel's buffer is undefined.

The **PARSE** command returns four-item nested vector, if successful. The first item is a simple character vector that is the value of the **WS** tag defined in the script file. If **WS** tag was not defined, the value of the *ws* item is an empty character vector.

The *vars* item of the result is a nested vector of simple character vectors. Each item of the *vars* is an executable APL expression that defines one CGI variable that corresponds to **NVAR** or to **CVAR** tag defined in the script file. The application should apply the *execute* (⍎) primitive to each item of the *vars* to define all thread CGI variables, at the application manager does during the thread initialization. The *vars* will be an empty character vector, if no **CVAR** and **NVAR** tags were defined in the script file.

The *exec* item of the result is a nested vector of simple character vectors. Each item of the *exec* is an executable APL expression that supports one **EXEC** tag defined in the script file. The *exec* will be an empty character vector, if no **EXEC** tags were defined in the script file.

The *fin* item of the result is a simple character vector that an executable APL expression that supports the **FINISH** tag defined in the script file. The *fin* will be an empty character vector, if the **FINISH** tag was not defined in the script file.

All system code needed to define CGI variables and to execute **EXEC** and **FINISH** tags is always located in the current thread. The application manager copies this code from the *ISAP* namespace, when a new thread is created. However, the application always should check the value of the *ws* item, because it may indicate a different application namespace, not which is being executed in the current thread. The name of the currently executed application tread can be obtained from the *_WS* environment variable (see section *Thread Environment Variables*). If values of *ws* and *_WS* do not match, the code that supports **EXEC** and **FINISH** tags may not work properly. The application may need to copy objects from the namespace, indicated by the *ws* item into the current thread. Also, note that this other application may use its latent expression to run (the value of the *_COM* global variable in the application namespace).

The purpose of the **PARSE** command is to switch the current script file and the current thread program. The HTML <FORM> tag can use only one ACTION parameter that defines the URL to use, when a user clicks one of SUBMIT-type controls on the CGI form. Sometimes it is necessary to use different URLs, when a user uses different SUMBIT controls. In many cases, such a situation can be resolved by creating another <FORM> element with the same set of hidden parameters. However, when both forms should use value(s) that user enters on the screen, the second form cannot be created. In such cases, APL programmers can use the **PARSE** command to effectively change the target script file, dependently on the submit button that has been used. See the example below.

If the application uses the **PARSE** command and executes a new thread program compiled by the command, it must abort the current thread program. If a new program is executing within an APL function that implements an **EXEC** tag, this defined function must set a new thread program (global variable *_COM*) and return scalar ¯1, as its explicit result. The use of the **PARSE** command directly by applications is not recommended. Applications should use the **_PARSE** function instead (see section *ISAP Functions Reference*). This function performs all described above actions needed to switch the script file.

Example:

The following example creates an Internet-based report on sales of certain item by MyCompany, Inc. within specified period of time. The user enters report dates and the item number and clicks button "Show Report". There is another SUMBIT button ("Find Item") on the form that invokes a helper application that allows finding the necessary item number using some search criteria. The search could be by item name, by sales volume, by the company's department and so on. When the user clicks the "Find Item" button, another application screen should appear to help a user to find the item. When the item number is found, user clicks "OK" button and the application shows the report screen again with the report dates restored and with the item number that user selected on the previous screen. The main application script file /isapdoc/report.xml is shown below:

```
<!--#ISAP WS=REPORT-->
<!--#INCLUDE /isapdoc/header.txt-->
<CENTER><H1>Sales Report</H2>
<!--#ISAP CVAR=ACTION NAME="ACTION" DEFAULT=""-->
<FORM METHOD=POST ACTION="/isapdoc/report.xml">
<INPUT TYPE=HIDDEN NAME="ALTFILE" VALUE="/isapdoc/items.xml">
<TABLE><TR><TD>Report start date:</TD>
<!--#ISAP CVAR=ST NAME="STDT" DEFAULT=""-->
<TD><INPUT TYPE=TEXT NAME="STDT" VALUE="<!--#ISAP EXEC="ST"-->"></TD>
<TD>Report end date:</TD>
<!--#ISAP CVAR=EN NAME="ENDT" DEFAULT=""-->
<TD><INPUT TYPE=TEXT NAME="ENDT" VALUE="<!--#ISAP EXEC="EN"-->"></TD>
</TR><TR><TD>Item number:</TD>
<!--#ISAP NVAR=ITEM NAME="ITEM" DEFAULT="0"-->
<TD>
<INPUT TYPE=TEXT NAME="ITEM" VALUE="<!--#ISAP EXEC="#.FMT ITEM"-->">
</TD>
<TD><INPUT TYPE=SUBMIT NAME="ACTION" VALUE="Find Item"></TD>
<TD></TD></TR></TABLE>
<INPUT TYPE=SUBMIT NAME="ACTION" VALUE="Show Report">
</FORM>
<!--#ISAP EXEC="REPORT ITEM ST EN"-->
<!--#INCLUDE /isapdoc/footer.txt-->
<!--#ISAP FINISH=""-->
```

The script file defines one **EXEC** tag that builds the report. Before the *REPORT* functions that takes *ITEM*, *ST*, and *EN* variables, as its arguments, is invoked, the application manager will define CGI variables that correspond to **CVAR** and **NVAR** tags found in the file. The hidden control ALTFILE specifies an alternate script file that should be used, when a user clicks the "Find Item" button. Note that the *REPORT* function will be executed when a user clicks either "Show Report" or "Find Item" buttons, as it is defined by the ACTION parameter of the <FORM> tag. Therefore, the *REPORT* function should check what button has been clicked and switch the script file, if needed. The *REPORT* function shown below uses ISAP's **_PARSE** function (see section *ISAP Functions Reference*):

```
     ∇ R←REPORT ARG;ITEM;STDT;ENDT;URL;PROG
[1]    ⍝ EXIT WITH EMPTY REPORT, IF NOTHING CLICKED
[2]    →('Show Report' 'Find Item'≡¨⊂ACTION)/REP,SWITCH ◊ →⍴R←''
[3]    ⍝ BUILDING THE REPORT
[4]    REP:ITEM STDT ENDT←ARG
[5]    ⍝ EXIT, IF ITEM IS 0
[6]    ⍞(0≠ITEM)/'R←''You have to enter an item number.'' ◊ →0'
[7]    ⍝ REPORT ERROR, IF DATES A WRONG
[8]    ⍞(0≠⍴R←CHECKDATE STDT)/'R←''Start date ivalid.'' ◊ →0'
[9]    ⍞(0≠⍴R←CHECKDATE ENDT)/'R←''End date invalid.'' ◊ →0'
[10]   R←PRODUCE_REPORT ITEM STDT ENDT ◊ →0
[11]   ⍝ SWITCH THE APPLICATION
[12]   SWITCH:R←_PARSE
     ∇
```

The following is an example of the alternate script file /isapdoc/items.xml. This file also contains hidden CGI control ALTFILE that allows returning back to the main application script file:

```
<!--#ISAP WS=ITEMS-->
<!--#INCLUDE /isapdoc/header.txt-->
<CENTER><H1>Select Item</H2>
<!--#ISAP NVAR=ITEM NAME="ITEM" DEFAULT="0"-->
<!--#ISAP CVAR=ST NAME="STDT" DEFAULT=""-->
<!--#ISAP CVAR=EN NAME="ENDT" DEFAULT=""-->
<!--#ISAP CVAR=ACTION NAME="ACTION" DEFAULT=""-->
<FORM METHOD=POST ACTION="/isapdoc/items.xml">
<INPUT TYPE=HIDDEN NAME="ALTFILE" VALUE="/isapdoc/report.xml">
<INPUT TYPE=HIDDEN NAME="STDT" VALUE="<!--#ISAP EXEC="ST"-->">
<INPUT TYPE=HIDDEN NAME="ENDT" VALUE="<!--#ISAP EXEC="EN"-->">
. . . . . .
<!--#ISAP EXEC="GETITEM"-->
<INPUT TYPE=SUBMIT NAME="IACTION" VALUE="Select">
<INPUT TYPE=SUBMIT NAME="IACTION" VALUE="Back">
</FORM>
<!--#INCLUDE /isapdoc/footer.txt-->
<!--#ISAP FINISH=""-->
```

We didn't show how exactly the product item is selected. The file may contain various CGI controls for this. There are a few important moments for us to show. Note how report start and end dates are saved in this script file. *ST* and *EN* CGI variables do not correspond to any visible controls on the screen. They just are being passes back and forth together with the HTTP request. This is how we keep their values in order to use them again on the report screen, when a user clicks the "Back" button. The *GETITEM* function that is called, when the **EXEC** tag is processed, works similar to the *REPORT* function shown above. It should analyze the value of the *IACTION* variable and return back to the reporting screen, when needed.

4.15. PARTITION

Format: *res←SAY 'PARTITION' chan tag [hdrs]*

Arguments:
chan - integer scalar: channel number.
tag - character vector: tag defined in the script file.
hdrs - character vector: optional HTTP headers.

Result: *res* - always 0.

Description:

The **PARTITION** command sends the beginning of the template file to the client via the channel *chan*. The template file must be loaded in the channel's buffer implicitly by ISAP, or explicitly by the **GETFILE** command. The *tag* parameter defines the part of the file to be sent. The command sends the content of the file starting from its beginning until the specified tag. The tag itself is not sent. The tag is removed from the file. The unsent part of the file remains in the channel's buffer. If the specified tag could not be found in the template file, nothing is sent.

The **PARTITION** command, when executed for the first time on channel *chan*, sends the HTTP status line "`200 OK`" and standard HTTP response headers: "`Content-Type: text/html`", "`Date:`", "`Last-Modified:`", "`Pragma: no-cache`". The "`Content-Length:`" header is not sent. The application may specify additional headers in the *hdrs* parameter. Each header must be terminated with the CR-LF pair, including the last header. If **HEADER** or **PARTITION** commands have been executed for this channel already, the *hdrs* parameter is ignored, even if it is present.

Because the **PARTITION** command does not send the "`Content-Length:`" HTTP header, the application must close the channel when it has finished processing, using the status code 0 (close channel and terminate TCP connection). Otherwise, the client won't be able to determine whether the transmission of the document is finished or not.

The purpose of the **PARTITION** command is to send the output document in parts. Sometimes, the application needs to create large parts of the document that would be too ineffective to keep in memory. In such cases the application may dump the final document simultaneously with its generation. The strategy here looks like follows:

- Replace small application tags with dynamic content using the **REPLACE** command.
- Send the beginning of the template file to the client using the **PARTITION** command.
- Send portions of the document as they are generated by the application using the **WRITE** command.
- Send the rest of the document remaining in the buffer using the **HTML** command.

The PARTITION command is executed asynchronously in a separate execution thread. See the database browser demo in the provided workspace.

Example:

The following example sends the template file to the client:

```
SAY 'PARTITION' 0 '<--!#TABLE-->' ''
SAY 'WRITE' 0 'A' TABLE
SAY 'HTML' 0
```

4.16. READ

Format: *data←SAY 'READ' chan [tt]*

Arguments: *chan* - integer scalar; channel number;
tt - character scalar; type of translation. See the description of the **CONVERT** command for explanations of conversion modes.
Allowed values are:
'N' - no translation;
'A' - translation from ANSI to Dyalog APL/W;
This parameter can be omitted. The *'A'* conversion is used in such a case.

Result: *data* - simple character vector; data read from the client.

Description:

This command reads data from a given channel *chan* using translation type *tt*. This command is used only when item 19 of the result of **GETINFO** is not 0. The result of the **READ** command is a simple character vector of arbitrary length. If **READ** returns an empty vector, no more data are pending.

If you expect more than 48Kb of character data from the client (a very rare case), which may contain escape sequences, you should not use the *'H'* conversion when you call the **GETINFO** command. Instead, use the *'N'* conversion for both **GETINFO** and **READ**, catenate the data, and use the **CONVERT** command with the *'H'* conversion on the data.

Example:

```
DATA←SAY 'READ' 3 'A'
```

4.17. REDIRECT

Format: *res←SAY 'REDIRECT' chan URL*

Arguments: *chan* - integer scalar: channel number;
URL - character vector: URL to be used.

Result: *res* - integer scalar: always 0.

Description:

This command instructs the client to use the specified URL, instead of the originally sent URL. The command sends a complete response 302. No further actions are required on the server side. The APL application ***must*** **CLOSE** the channel after using the **REDIRECT** command. Upon receiving such a response, the client's browser will automatically call the URL specified by the *URL* parameter. The specified URL may or may not be on the same server. If the specified URL refers to an object on the same sever, it should not contain the protocol information.

Can be used for effective switches between different servers.

If an application uses the **REDIRECT** command inside a defined function that implements an **EXEC** tag, the function should return 0 as its explicit result. The application manager will terminate the current thread program and will close the channel. If an application uses the **REDIRECT** command inside thread latent expression, the application should explicitly reset thread program by setting the *_COM* global variable to an empty vector.

Example:

Redirection to URL on a different server:

```
SAY 'REDIRECT' CHAN 'http://serv.other.net/welcome.htm'
```

Redirection to URL on the same server:

```
SAY 'REDIRECT' CHAN '/isap/home.htm'
```

4.18. REPLACE

Format: *res←SAY 'REPLACE' chan pat str*

Arguments:
chan - integer scalar: channel number.
pat - character vector: pattern to replace.
str - character vector, or an empty vector, or scalar 0: replacement string.

Result: *res* - integer scalar: always 0.

Description:

The **REPLACE** command replaces the pattern *pat* with the string *str* in the template file that is currently loaded in the channel's buffer. The channel is specified by the *chan* argument. The template file could be loaded implicitly by ISAP as the result of processing of the HTTP request that refers to the file registered as an ISAP script file (default extension ".xml"), or it could be loaded explicitly by the application using the **GETFILE** command on the same channel. All occurrences of the pattern *pat*, if any, will be replaced. The pattern vector cannot be empty. The replacement string can be empty.

If the *str* parameter is scalar 0, this command is ignored.

Applications should avoid using **REPLACE** command. Instead, programmers should use ISAP scripting facilities (see chapter *ISAP Tag Reference*) to create dynamic elements in the template file, when it is acceptable. The **REPLACE** command should be used to process application-specific tags.

Example:

The following is an example of the template HTML file c:\InetPub\doc\ipdemo.htm:

```
<HTML>
<HEAD><TITLE>Template HTML File Usage Demo</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000" LINK="#0000FF" VLINK="#551A8B">
<FONT FACE="COMIC SANS MS,ARIAL">
<CENTER>
<H1>Client IP Address Demo</H1>
<P>
Your IP address is: <FONT COLOR="#FF0000"><!--#IP--></FONT>.
</BODY></HTML>
```

In the following APL code example reads the template file, obtains the client's IP address as a character vector as the result of the *GETIPADDR* defined function, replaces tag <!--#IP--> with the actual value and sends the result HTML document to the client:

```
SAY 'GETFILE' CHAN 'C:\INETPUB\DOC\IPDEMO.HTM'
SAY 'REPLACE' CHAN '<!--#IP-->' GETIPADDR
SAY 'HTML' CHAN
```

The **GETFILE** command used in the above example wouldn't be needed if the template file had the extension "xml". ISAP automatically loads such files when clients request them, before sending the notification message to the APL application. See also descriptions of **HTML**, **MAPURL**, **FREPLACE** and **GETFILE** commands. A better solution for the above example would be to use the **EXEC** tag to insert the IP address into the document.

4.19. SENDURL

Format: `res←SAY 'SENDURL' chan URL`

Arguments: `chan` - integer scalar: channel number;
`URL` - character vector: URL to be used.

Result: `res` - integer scalar: always 0.

Description:

This command instructs the IIS to use the specified URL, as if the client requested it, instead of the originally requested URL. No further actions are required on the server side. The APL application ***must*** **CLOSE** the channel after using the **SENDURL** command. Internet Server will automatically send the specified URL to the client. The specified URL ***must*** be on the same server, it ***must not*** contain the server address and protocol information. Thus, it always begins with the "/" character.

This command can be used for effective error processing and switching between applications.

If an application uses the **SENDURL** command inside a defined function that implements an **EXEC** tag, the function should return 0 as its explicit result. The application manager will terminate the current thread program and will close the channel. If an application uses the **SENDURL** command inside thread latent expression, the application should explicitly reset thread program by setting the `_COM` global variable to an empty vector.

Example:

```
SAY 'SENDURL' CHAN '/isap/ferror.htm'
```

4.20. TAGS

Format: *tags←SAY 'TAGS' chan tbeg tend*

Arguments: *chan* - integer scalar: channel number.
tbeg - character vector: pattern that begins the tag.
tend - character vector: pattern that ends the tag.

Result: *tags* - nested vector of simple character vectors or an empty vector.

Description:

The **TAGS** command scans the currently loaded template file on channel *chan* and returns all character strings that start with the pattern *tbeg* and end with the pattern *tend*, not including starting and ending patterns themselves. All strings found are returned in order of their appearance in the file. The template file could be loaded implicitly by ISAP as the result of processing of the HTTP request that refers the file of the type registered as ISAP script file (default extension ".xml"), or it could be loaded explicitly by the application using the **GETFILE** command on the same channel. Both pattern vectors cannot be empty.

This command implements an effective search for specific text fragments. Combined with the **REPLACE** and **FREPLACE** commands, the **TAGS** command allows processing of the template file in order to create the final HTML document that is sent to the client using the **HTML** command. Users can implement their own script interpreters to enhance the standard ISAP script processor.

Example:

The following template file has been loaded into the buffer on channel *chan*:

```
<HTML>
<HEAD><TITLE>INCLUDE Tags Demo</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000" LINK="#0000FF" VLINK="#551A8B">
<FONT FACE="COMIC SANS MS, ARIAL">
<!--#INCL /doc/include1.txt-->
<P>
Your IP address is: <FONT COLOR="#FF0000"><!--#ISAP IP--></FONT>.
<!--#INCL /doc/include2.txt-->
</BODY></HTML>
```

Find all custom tags in the above file:

```
      ⍴A←SAY 'TAGS' chan '<!--#' '-->'
3
      A
 INCL /doc/include1.txt  ISAP IP  INCL /doc/include2.txt
```

Find all `#INCL` tags in the above file:

```
      ⍴A←SAY 'TAGS' chan '<!--#INCL ' '-->'
2
      A
 /doc/include1.txt  /doc/include2.txt
```

Find all `#ISAP` tags in the above file:

```
      ⍴A←SAY 'TAGS' chan '<!--#ISAP ' '-->'
1
      A
 IP
```

4.21. TIME

Format: *time*←*SAY* *'TIME'* [*offset*]

Arguments: *offset* - integer scalar: time offset in seconds. If omitted, the value 0 is used.

Result: *time* - character vector of length 29: HTTP date/time representation.

Description:

This command returns a properly formatted date/time string. If the *offset* parameter is 0 or omitted, the **TIME** command returns the current time. Otherwise, it returns time shifted on specified offset. The offset value should be specified in seconds and can be negative. This command can be used for creating HTTP headers that require date/time information.

There is a more efficient way to get time in the HTTP format. See chapter *ISAP Functions Reference.*

Example:

The following examples use the **TIME** command to obtain the date/time string in the HTTP format:

```
      ⎕←SAY 'TIME'
Fri, 30 May 1997 19:27:59 GMT
      ⎕←SAY 'TIME' 3600
Fri, 30 May 1997 20:29:46 GMT
      ⎕←SAY 'TIME' ¯3600
Fri, 30 May 1997 18:30:39 GMT
```

4.22. WRITE

Format: `len←SAY 'WRITE' chan tt data`

Arguments:
- `chan` - integer scalar: channel number;
- `tt` - character scalar: translation type. Can be one of the following:
  - `'N'` - no translation;
  - `'A'` - Dyalog APL to ANSI translation;
- `data` - character vector: data to be sent.

Result: `len` - integer scalar: number of bytes written to the output buffer.

Description:

This command is used to send the result document to the client. APL applications may prepare the complete HTTP response that includes the status line, headers, and the body, as a character vector and send it in a single **WRITE** call. You also can take advantage of using the **HEADER** command (recommended) that simplifies the procedure of sending headers data. It is especially useful when you are sending binary data (images, for example). Any number of **WRITE** commands can be used during the processing of an HTTP request.

The **WRITE** command sends `data` to the output buffer using the specified translation type, starts the asynchronous transmission procedure on channel `chan`, and exits. The amount of time needed to finish the transmission depends on server performance and the connection speed. Because the **WRITE** command performs the transmission asynchronously, it returns immediately. The return value `len` shows number of bytes submitted for the transfer (it always will be the length of `data`).

As soon as application starts to use the **WRITE** command, it cannot send any other commands to a given channel except **WRITE** and **CLOSE**. The **WRITE** command accepts data of any length. So, in order to reach maximum performance, single-step applications (see chapter *Developing ISAP Applications*) should prepare the complete response body, if they can, and send all data in one **WRITE** call. If the application prepares a long output document in more than one call of the application manager, it is, probably, a good idea to use more than one **WRITE** command on each step of execution.

Example:

```
      L←SAY 'WRITE' 3 'A' 'SOME DATA'
      L
9
```

4.23. 3DBAR

Format: `res←SAY '3DBAR' arg`

Arguments: `arg` - 11-element nested vector.

Result: `res` - integer scalar: always 0.

Description:

This command asynchronously draws a 3D bar chart using Olectra Chart 5.0 (file olch3d32.dll located in the /ISAP virtual directory of the IIS). After issuing this command the application must close the channel using the **CLOSE** command. Bar chart creation and image transfer are done in the server process, not in the APL process. This significantly increases the performance of graphical applications.

`arg` should be an 11-element nested vector according to the specifications below. You should use the sample Web site and the sample APL workspace (*Server-Side Graphics* page) to experiment with the parameter values.

`1⊃arg` - integer scalar, ISAP channel number.

`2⊃arg` - 7-element integer vector, chart appearance:

- `(2⊃arg)[1]` - integer between 1 and 15, chart type. There are 15 possible chart types which are combinations of four flags: 1 - zones; 2 - contours; 4 - shaded; 8 - mesh. The chart type is constructed by adding the desired flags. You can experiment with different combinations of these flags using the supplied demo Web site and the APL workspace.
- `(2⊃arg)[2]` - chart width in pixels.
- `(2⊃arg)[3]` - chart height in pixels.
- `(2⊃arg)[4]` - can be 0 or 1. If this is 1, the chart will be transparent. If it is 0, the chart will be drawn on white background.
- `(2⊃arg)[5]` - can be 0 or 1. If it is 1, X-axis grid lines will be drawn.
- `(2⊃arg)[6]` - can be 0 or 1. If it is 1, Y-axis grid lines will be drawn.
- `(2⊃arg)[7]` - must be between 1 and 1000. Defines relative size of the axes font.

`3⊃arg` - 5-element numeric vector, chart scaling:

- `(3⊃arg)[1]` - X axis scaling;
- `(3⊃arg)[2]` - Y axis scaling;
- `(3⊃arg)[3]` - Z axis scaling;
- `(3⊃arg)[4]` - bar spacing along X-axis. Must be number between 0 and 100. For example, if this is 50, bars and spaces between bars will be the same along X-axis.
- `(3⊃arg)[5]` - bar spacing along Y-axis (see above).

The following three vectors define the chart's grid and chart's values for the corresponding (x,y) pairs:

`4⊃arg` - numeric vector; x-values.

`5⊃arg` - numeric vector; y-values.

`6⊃arg` - numeric matrix; z-values. This matrix must have as many rows as the length of `4⊃arg` and as many columns, as the length of `5⊃arg`.

`7⊃arg` - numeric matrix; bars colors. Scalar can be used instead (will be treated as a matrix with one row and one column). This matrix must have the same shape as the shape of `6⊃arg`. Any dimension may have length 1. In this case, the matrix will be expanded along this dimension. For example, if `6⊃arg` has shape 5 4, and `7⊃arg` has shape 1 4, the first row will be repeated 5 times before using. Each element of this matrix represents the color to be used for drawing of the corresponding bar. The color is defined as an RGB value calculated as the result of the expression: `256⊥B G R` - where *R*, *G*, and *B* are integers between 0 and 255, and specify the intensity of red, green, and blue.

`8⊃arg` - nested vector of simple character vectors, or an empty vector. Any of these simple vectors can be empty. This defines labels for the X-axis. If the parameter is empty, x-values will be used for X-axis annotation.

`9⊃arg` - nested vector of simple character vectors, or an empty vector. Any of these simple vectors can be empty. This defines labels for the Y-axis. If the parameter is empty, y-values will be used for Y-axis annotation.

`10⊃arg` - nested vector of 3 simple character vectors. Any of these simple vectors can be empty. This defines titles for X, Y, and Z axes.

`11⊃arg` - nested vector, legend specification. The legend can be drawn only when *contours* and *zones* flags are used in the chart type (see `2⊃arg`). For other chart types this is ignored.

`(11⊃arg)[1]` - integer scalar 0 or 1, legend presence.

`(11⊃arg)[2]` - simple character vector, legend font specification. This defines the font to be used in the legend. The format is "*font-name,font-size,font-type*". For example: `'MS Sans Serif,8,normal'`. If it is an empty vector, the default font is used.

Example:

The following example draws a bar chart with all red bars. The default axes labeling is used.

```
X←1 2 3 4
Y←1 2 3 4 5 6
Z←4 6⍴24?24
TYPE←8+4
COLS←255  ⍝ ALL BARS ARE RED
TIT←'X axis' 'Y axis' 'Z axis'
LEGEND←0 ''
ARG←CHAN(TYPE,500 300 0 1 1 70)(1 1 1 60 60)X Y Z COLS '' '' TIT LEGEND
SAY '3DBAR' ARG
```

See also the *CHDEMO* namespace in the supplied sample workspace.

4.24. 3DCHART

Format: `res←SAY '3DCHART' arg`

Arguments: `arg` - 10-element nested vector;

Result: `res` - integer scalar; always 0.

Description:

This command asynchronously draws a 3D surface chart using Olectra Chart 5.0 (file olch3d32.dll located in the /ISAP virtual directory of IIS). After issuing this command the application must close the channel using the **CLOSE** command. Surface chart creation and the image transfer is done in the server process, not in the APL process. This significantly increases the performance of graphical applications.

`arg` should be 10-element nested vector according to the specifications below. You should use the sample Web site and the sample APL workspace (*Server-Side Graphics* page) to experiment with the parameter values.

`1⊃arg` - integer scalar, ISAP channel number.

`2⊃arg` - 8-element integer vector, chart appearance:

- `(2⊃arg)[1]` - integer between 1 and 15, chart type. There are 15 possible chart types which are combinations of four flags: 1 - zones; 2 - contours; 4 - shaded; 8 - mesh. The chart type is constructed by adding the desired flags. You can experiment with different combinations of these flags using the supplied demo Web site and the APL workspace.
- `(2⊃arg)[2]` - chart width in pixels.
- `(2⊃arg)[3]` - chart height in pixels.
- `(2⊃arg)[4]` - can be 0 or 1. If this is 1, the chart will be transparent. If it is 0, the chart will be drawn on white background.
- `(2⊃arg)[5]` - can be 0 or 1. If it is 1, X-axis grid lines will be drawn.
- `(2⊃arg)[6]` - can be 0 or 1. If it is 1, Y-axis grid lines will be drawn.
- `(2⊃arg)[7]` - must be between 1 and 1000. Defines relative size of the axes font.
- `(2⊃arg)[8]` - can be 0 or 1. If it is 1, the "solid surface" will be drawn.

`3⊃arg` - 3-element numeric vector, chart scaling:

- `(3⊃arg)[1]` - X axis scaling;
- `(3⊃arg)[2]` - Y axis scaling;
- `(3⊃arg)[3]` - Z axis scaling;

The following three vectors define X-Y grid and surface values for the corresponding (x,y) pairs:

`4⊃arg` - numeric vector; x-values.

`5⊃arg` - numeric vector; y-values.

`6⊃arg` - numeric matrix; z-values. This matrix must have as many rows as the length of `4⊃arg` and as many columns, as the length of `5⊃arg`.

`7⊃arg` - nested vector of simple character vectors, or an empty vector. Any of these simple vectors can be empty. This defines labels for the X-axis. If the parameter is empty, x-values will be used for X-axis annotation.

`8⊃arg` - nested vector of simple character vectors, or an empty vector. Any of these simple vectors can be empty. This defines labels for the Y-axis. If the parameter is empty, y-values will be used for Y-axis annotation.

`9⊃arg` - nested vector of 3 simple character vectors. Any of these simple vectors can be empty. This defines titles for X, Y, and Z axes.

`10⊃arg` - nested vector, legend specification. The legend can be drawn only when *contours* and *zones* flags are used in the chart type (see `2⊃arg`). For other chart types this is ignored.

`(11⊃arg)[1]` - integer scalar 0 or 1, legend presence.

`(11⊃arg)[2]` - simple character vector, legend font specification. This defines the font to be used in a legend. The format is "*font-name,font-size,font-type*". For example: `'MS Sans Serif,8,normal'`. If it is an empty vector, the default font is used.

Example:

The following example draws a surface chart. The default axes labeling is used. Note Z-axis scaling.

```
X←1 2 3 4
Y←1 2 3 4 5 6
Z←4 6⍴24?24
TYPE←8+4
TIT←'X axis' 'Y axis' 'Z axis'
LEGEND←0 ''
ARG←CHAN(TYPE,500 300 0 1 1 70 0)(1 1 2)X Y Z '' '' TIT LEGEND
SAY '3DBAR' ARG
```

See also the *CHDEMO* namespace in the supplied sample workspace.

5. ISAP Functions Reference

The Internet Server Auxiliary Processor application development environment includes a set of functions, most of which are supported by ISAPG.DLL located in the same directory where the application manager workspace resides. Some functions are purely exported from this DLL. They are attached to the APL workspace by the *ISAPStart* function when the application manager starts. Other functions are APL defined functions that usually call functions imported from the ISAPG.DLL. Functions imported from ISAPG.DLL are deleted at shutdown by the *ISAPEnd* function. *QNAStart* also attaches some useful functions from Windows DLLs.

WARNING

The ISAPG.DLL is provided as a part of the ISAP programming product. It cannot be used for any other purposes without a written permission from Lingo Allegro USA, Inc.

Some functions in the ISAPG.DLL implement the Compuserve graphics interchange format (GIF), which uses LZW compression. Use of this software for providing LZW capability for any purpose is not authorized unless the user first enters into a license agreement with Unisys under U.S. Patent No. 4,558,302 and foreign counterparts. See page 4 of this document.

If you need to attach other functions from other DLLs, you have to write two functions, similar to *QNAStart* and *QNAEnd*, and to place them at the end of these functions.

There are three places in the application manager workspace where ISAP functions can be found: root namespace (*#*), *ISAP* namespace, and *ISAPCLI* namespace. All ISAP general-purpose defined functions are located in the *ISAP* namespace. All functions imported from ISAPG.DLL are located in the root namespace.

When using the ISAP functions, applications must use absolute namespace reference (relative to the root namespace) to ensure that the applications are able to run from any other namespace. Thus, users must use fully qualified names, including the namespace names, when referring to these functions in their applications.

ISAPCLI namespace contains set of functions that are copied to all request namespaces, when they are created. These functions must be present in all client's namespaces, because the application manager calls them, when executing scripts. You can place your own functions, which you need to be available in all request namespaces, if you want. Usually, you don't need to do this, unless a function must be executed in the context of the current namespace.

For distribution and update purposes, all ISAP functions are located in the *ISAP* namespace. Some of them, which should be executed in the root namespace, are copied to the root by the *SETGLOBALS* function, when ISAP starts. Users should not modify, delete, or create new functions in the *ISAP* namespace. This namespace might be different in the next version of ISAP. In the descriptions below, the locations of the functions are given at run-time. Programmers should use function definitions, as they are given in this document, even some of the functions can not be found in the supplied workspace, when it is not running.

You can delete all objects from the *ISERV* workspace, except root, *ISAP*, and *ISAPCLI* namespaces.

The following functions are supported by ISAP. Detailed descriptions are given later in this chapter. All function names reflect function locations. Function descriptions below use Dyalog APL syntax coding, where brackets "{" and "}" are used to indicate optional arguments and "shy" results. Some functions must run in the current application thread to work properly. Such functions are copied to the application thread when it is created. These functions must be referenced by their names only, without any namespace specification.

Functions located in the root namespace:

#.DrawBitmap	draws a bitmap on specified device context.
#.GIFCreateDC	creates a picture context that can be converted to a GIF or to an XBM image.
#.GIFDeleteDC	deletes a picture context.
#.ISAPEnd	terminates connection between ISAP and APL.
#.ISAPStart	establishes connection between ISAP and APL.
#.SAY	sends commands to ISAP.
#.User	logs APL task under specified account.
#.Δ OUT	logs a debugging message into the application log file.

Functions located in the *ISAP* namespace:

#.ISAP.CreateLogFile	creates or opens application manager log file.
#.ISAP.DEB	deletes leading, trailing, and embedded blanks.
#.ISAP.DRAW_GIF	creates a GIF image from the specified picture context.
#.ISAP.DRAW_GIFA	asynchronously creates a GIF image from the specified picture context.
#.ISAP.DRAW_GIFF	creates a GIF image file from the specified picture context.
#.ISAP.DRAW_PNG	creates a PNG image from the specified picture context.
#.ISAP.DRAW_PNGA	asynchronously creates a PNG image from the specified picture context.
#.ISAP.DRAW_PNGF	creates a PNG image file from the specified picture context.
#.ISAP.DRAW_XBM	creates an XBM image from the specified picture context.
#.ISAP.GetBitmap	creates a bitmap object from a file.
#.ISAP.GET_COOKIE	retrieves the value of a cookie.
#.ISAP.GETHTML	loads specified template file.
#.ISAP.GETENTRY	retrieves entry value from initialization file.
#.ISAP.IMAGE	asynchronously creates a GIF image from APL metafile object.
#.ISAP.IMAGEF	creates a GIF image file from APL metafile object.
#.ISAP.IMAGEX	creates x-xbitmap image from APL metafile object.
#.ISAP.IMAGEXF	creates x-xbitmap image file from APL metafile object.
#.ISAP.PIMAGE	asynchronously creates a PNG image from APL metafile object.
#.ISAP.PIMAGEF	creates a PNG image file from APL metafile object.
#.ISAP.PUTENTRY	writes entry value in initialization file.
#.ISAP.repl	effective string search and replacement.
#.ISAP.RUNPROGR	starts external task.
#.ISAP.TIME	returns time relative to the current time in the HTTP format.
#.ISAP.TOLOWER	converts character strings to lower case.
#.ISAP.TOUPPER	converts character strings to upper case.

#.ISAP.TRBLANKS removes leading and trailing blanks from a character vector.

#.ISAP._PARAMETER retrieves the value of a CGI variable.

#.ISAP.DELCOOKIE returns Set-Cookie HTTP header for a cookie to be deleted.

#.ISAP.SETCOOKIE returns Set-Cookie HTTP header for the cookie to be set.

Functions stored in the *ISAPCLI* namespace and copied in the current thread (namespace):

_PARSE loads a new script file, parses it, and starts the execution of a new thread program.

_CVAR supports CVAR tags (not documented).

_NVAR supports NVAR tags (not documented).

_EXEC supports EXEC tags (not documented).

_COOK supports COOKIE tags (not documented).

if supports "if" expression to be used in **EXEC** tags.

in supports "in" expression to be used in **EXEC** tags.

is supports "is" expression to be used in **EXEC** tags.

takes supports assignment operation to be used in **EXEC** tags.

selected returns string " selected" to be used in **EXEC** tags.

checked returns string " checked" to be used in **EXEC** tags.

5.1. #.DrawBitmap

Syntax: `res←#.DrawBitmap hDC hBitmap x y`

Arguments:

`hDC`	-	picture device context handle returned by the `GIFCreateDC` function.
`hBitmap`	-	bitmap handle returned by the `GetBitmap` function.
`x`	-	x-coordinate on the device context.
`y`	-	y-coordinate on the device context.

Result: `res` - simple numeric scalar: 0, if the function is successful, and 1 - otherwise.

Description:

The `DrawBitmap` function draws a bitmap, defined by `hBitmap` on the device context defined by `hDC`. Upper left corner of the bitmap is placed at the position of the device context defined by `x` and `y` values. This function allows assembling complex images from pre-defined bitmaps.

Example:

The following defined function `BMPtoGIF` takes the ISAP channel number as its left argument, and a bitmap file name as its right argument. It creates a GIF image from the bitmap file and sends it to the client.

```
     ∇ CHAN BMPtoGIF FILE;hBitmap;BmpSize;hDC;R
[1]    ⍝ READ BITMAP FROM THE FILE
[2]    hBitmap BmpSize←#.ISAP.GetBitmap FILE
[3]    ⍝ CREATE PICTURE DEVICE CONTEXT OF BITMAP SIZE
[4]    hDC←#.GIFCreateDC BmpSize
[5]    ⍝ DRAW BITMAP ON THE DEVICE CONTEXT
[6]    R←#.DrawBitmap hDC hBitmap 0 0
[7]    ⍝ CREATE AND SEND HTTP RESPONSE WITH THE GIF IMAGE
[8]    R←#.ISAP.DRAW_GIF hDC '' CHAN
[9]    ⍝ DELETE PICTURE DEVICE CONTEXT
[10]   R←#.GIFDeleteDC hDC
[11]   ⍝ DELETE BITMAP
[12]   R←#.DeleteObject hBitmap
     ∇
```

5.2. #.GIFCreateDC

Syntax: `hDC←#.GIFCreateDC width height`

Arguments:

`width`	-	width of the picture device context in pixels.
`height`	-	height of the picture device context in pixels.

Result: `hDC` - picture device context handle; will be 0, if device context could not be created.

Description:

The `#.GIFCreateDC` function creates a picture device context of the specified size. The picture device context behaves as a normal logical Windows device context. It can be used for any graphical operations. The content of this device context can be converted into an HTTP response containing a GIF or x-xbitmap image using the `#.ISAP.DRAW_GIF` and `#.ISAP.DRAW_XBM` functions. The device context must be deleted by the `#.GIFDeleteDC` function when it is no longer needed. Maximum number of picture device contexts that can exist at the same time is 256.

Example:

See example in the description of the `#.DrawBitmap` function.

5.3. #.GIFDeleteDC

Syntax: *res←#.GIFDeleteDC hDC*

Arguments: *hDC* - picture device context handle returned by the *#.GIFCreateDC* function.

Result: *res* - non-zero, if the device context was successfully deleted.

Description:

The *#.GIFDeleteDC* function deletes a picture device context that was created by a previous call to the *#.GIFCreateDC* function. Programmers should keep the number of created picture device contexts as low as possible to save resources on the server.

Example:

See example in the description of the *#.DrawBitmap* function.

5.4. #.ISAPEnd

Syntax: *{res}←#.ISAPEnd*

Arguments: none.

Result: *res* - integer scalar: 1, if the function is successful, 0 - otherwise.

Description:

This function must be executed by the APL application manager before it exits. If the application manager does not execute this function, it might prevent a proper shutdown of the Internet server. The *ISAPEnd* function also destroys all functions imported from the ISAPG.DLL. The standard application manager executes this function during execution of the *END* function.

5.5. #.ISAPStart

Syntax: *res←#.ISAPStart*

Arguments: none.

Result: *res* - integer scalar: 1, if the function is successful, 0 - otherwise.

Description:

The *ISAPStart* function establishes a connection between an application workspace and ISAP. It returns 1, if it is successful. If the function returns 0, the application must terminate. The application has about 30 seconds from the moment when the APL workspace starts executing to establish the connection with ISAP and to send the **NOTIFY** command. The APL interpreter will be terminated by the Internet server if the application fails to do so. The application manager executes the *ISAPStart* function when it executes the *SHARE* function. The function also attaches external functions from ISAPG.DLL

5.6. #.SAY

Syntax: `{res}←#.SAY command`

Arguments: `command` - an APL array.

Result: `res` - an APL array.

Description:

The `SAY` function sends commands from the APL application to ISAP. The `command` argument is command-dependent. See chapter *ISAP Command Reference.* The result of the function is the command's result that is also command-dependent. The function will generate error 667 if an error occurs during the processing of the command. The `⎕DM` system variable will contain the diagnostic message. The application manager will close the ISAP channel if this error is untrapped by the application.

The application must execute the `ISAPStart` function to be able to communicate with ISAP.

5.7. #.User

Syntax: `res←#.User acc dom pwd`

Arguments: `acc` - simple character vector - Windows NT account.
`dom` - simple character vector - domain or server name, where the account is stored.
`pwd` - simple character vector - password.

Result: `res` - integer scalar: 1, if the function is successful, 0 - otherwise.

Description:

When ISAP loads Dyalog APL, the APL task is executing under the LocalSystem account. If the application requires access to resources that are not accessible under this account (usually, network resources), you can use this command to re-log under a different account. The `dom` parameter can be `'.'`, in which case only the computer's local account database is searched. Note that the `User` function does not change the content of the registry. This means that the content of CURRENT_USER will not be changed. Moreover, because ISAP runs as a part of NT service, there may be no logged user at all. If you need to access registry settings when working with ISAP, use the CURRENT_MACHINE registry to store necessary information. For instance, if you need to connect to an ODBC data source, define this source as a Machine Data Source, not as a User Data Source.

Example:

The following examples use the `#.User` command to log the APL application under account *ANDREI*, valid in *LINGO* domain, using password *Password*:

```
      #.User 'ANDREI' 'LINGO' 'Password'
1
```

5.8. #.ISAP.CreateLogFile

Syntax: `tn←#.ISAP.CreateLogFile FileName`

Arguments: `FileName` - fully qualified file name.

Result: `tn` - Integer: file tie number, or 0, if file cannot be opened or created.

Description:

The `#.ISAP.CreateLogFile` function opens the specified file in text mode. If the file does not exist, it will be created. The returned tie number can be used by applications to log records in the application log.

Example:

```
      #.ISAP.CreateLogFile 'c:\LogFiles\isaplog.txt'
¯1
```

5.9. #.ISAP.DRAW_GIF

Syntax: `#.ISAP.DRAW_GIF hDC rgb chan [head]`

Arguments:
`hDC` - picture device context handle returned by the `#.GIFCreateDC` function.
`rgb` - three element integer vector, or empty vector: transparent color.
`chan` - integer scalar: channel number.
`head` - optional simple character vector: additional HTTP headers.

Result: None.

Description:

The `DRAW_GIF` function converts a picture drawn on the picture device context `hDC` into an HTTP response that contains that picture in GIF format and sends this response to the client, identified by the channel number `chan`. The `rgb` parameter is a three-element integer vector. Each element must be between 0 and 255 and defines intensity of red, green, or blue color. All together the values of `rgb` specify the transparent color of the resulting GIF image. If `rgb` is empty, the image will not be transparent.

The optional `header` argument specifies additional HTTP headers to be included in the result. If the `header` is not present, the function creates standard HTTP headers: "Content-Type", "Content-Length", "Date", "Last-Modified", and "Pragma: no-cache". The application may specify additional headers, if it needs to, like "Cookie". Each header must be terminated with CR-LF pair (`⎕AV[4 3]`).

The `DRAW_GIF` function executes synchronously. It does not return until the response is placed to the output queue on the server. The application must close the channel after the function returns. The `DRAW_GIF` function can be applied to the same picture device context `hDC` more than once (during processing other requests).

The use of this function requires a license from Unisys.

Example:

See example in the description of the `#.DrawBitmap` function.

5.10. #.ISAP.DRAW_GIFA

Syntax: `#.ISAP.DRAW_GIFA hDC rgb chan [head]`

Arguments:
- `hDC` - picture device context handle returned by the `#.GIFCreateDC` function.
- `rgb` - three element integer vector, or empty vector: transparent color.
- `chan` - integer scalar: channel number.
- `head` - optional simple character vector: additional HTTP headers.

Result: None.

Description:

The `DRAW_GIFA` function converts a picture drawn on the picture device context `hDC` into an HTTP response that contains that picture in GIF format and sends this response to the client, identified by the channel number `chan`. The `rgb` parameter is a three-element integer vector. Each element must be between 0 and 255 and defines intensity of red, green, or blue color. All together the values of `rgb` specify the transparent color of the resulting GIF image. If `rgb` is empty, the image will not be transparent.

The optional `header` argument specifies additional HTTP headers to be included in the result. If the `header` is not present, the function creates standard HTTP headers: "Content-Type", "Content-Length", "Date", "Last-Modified", and "Pragma: no-cache". The application may specify additional headers, if it needs to, like "Cookie". Each header must be terminated with CR-LF pair (`⎕AV[4 3]`).

The `DRAW_GIFA` function executes asynchronously in a separate Windows NT thread. It returns to the APL immediately, after the thread is started. The application must close the channel `chan` after the function returns. The `DRAW_GIFA` function destroys the picture device context `hDC`, after the response has been sent to the client. Thus, the application **must** de-select all GDI objects that were selected in this device context before it calls the function. Application **must not** use `hDC` handle after the call.

In all places, when an APL application does not need to re-use the picture device context, it should use `DRAW_GIFA` function instead of `DRAW_GIF` function. This will improve the overall server performance, especially on multiprocessor computers. The use of this function requires a license from Unisys.

Example:

See example in the description of the `#.DrawBitmap` function.

5.11. #.ISAP.DRAW_GIFF

Syntax: `{size}←file #.ISAP.DRAW_GIFF hDC rgb`

Arguments:
- `hDC` - picture device context handle returned by the `#.GIFCreateDC` function.
- `rgb` - three element integer vector, or empty vector: transparent color.
- `file` - simple character vector: fully qualified output file name.

Result:
- `size` - simple one-element integer vector: the size of the created file in bytes.

Description:

The `DRAW_GIFF` function converts a picture drawn on the picture device context `hDC` into picture in GIF format and saves the picture in the specified `file`. The `rgb` parameter is a three-element integer vector. Each element must be between 0 and 255 and defines intensity of red, green, or blue color. All together the values of `rgb` specify the transparent color of the resulting GIF image. If `rgb` is empty, the image will not be transparent.

The `DRAW_GIFF` function executes synchronously. It does not return until the image file is created. The function can be applied to the same picture device context `hDC` more than once, if needed.

The use of this function requires a license from Unisys.

5.12. #.ISAP.DRAW_PNG

Syntax: `#.ISAP.DRAW_PNG hDC rgb chan [head]`

Arguments:

`hDC`	- picture device context handle returned by the `#.GIFCreateDC` function.
`rgb`	- three element integer vector, or empty vector: transparent color.
`chan`	- integer scalar: channel number.
`head`	- optional simple character vector: additional HTTP headers.

Result: None.

Description:

The `DRAW_PNG` function converts a picture drawn on the picture device context `hDC` into an HTTP response that contains that picture in PNG format and sends this response to the client, identified by the channel number `chan`. The `rgb` parameter is a three-element integer vector. Each element must be between 0 and 255 and defines intensity of red, green, or blue color. All together the values of `rgb` specify the transparent color of the resulting PNG image. If `rgb` is empty, the image will not be transparent.

The optional `header` argument specifies additional HTTP headers to be included in the result. If the `header` is not present, the function creates standard HTTP headers: "Content-Type", "Content-Length", "Date", "Last-Modified", and "Pragma: no-cache". The application may specify additional headers, if it needs to, like "Cookie". Each header must be terminated with CR-LF pair (`⎕AV[4 3]`).

The `DRAW_PNG` function executes synchronously. It does not return until the response is placed to the output queue on the server. The application must close the channel after the function returns. The `DRAW_PNG` function can be applied to the same picture device context `hDC` more than once (during processing other requests).

Example:

The following defined function `BMPtoPNG` takes the ISAP channel number as its left argument, and a bitmap file name as its right argument. It creates a PNG image from the bitmap file and sends it to the client.

```
     ∇ CHAN BMPtoPNG FILE;hBitmap;BmpSize;hDC;R
[1]    ⍝ READ BITMAP FROM THE FILE
[2]    hBitmap BmpSize←#.ISAP.GetBitmap FILE
[3]    ⍝ CREATE PICTURE DEVICE CONTEXT OF BITMAP SIZE
[4]    hDC←#.GIFCreateDC BmpSize
[5]    ⍝ DRAW BITMAP ON THE DEVICE CONTEXT
[6]    R←#.DrawBitmap hDC hBitmap 0 0
[7]    ⍝ CREATE AND SEND HTTP RESPONSE WITH THE PNG IMAGE
[8]    R←#.ISAP.DRAW_PNG hDC '' CHAN
[9]    ⍝ DELETE PICTURE DEVICE CONTEXT
[10]   R←#.GIFDeleteDC hDC
[11]   ⍝ DELETE BITMAP
[12]   R←#.DeleteObject hBitmap
     ∇
```

5.13. #.ISAP.DRAW_PNGA

Syntax: `#.ISAP.DRAW_PNGA hDC rgb chan [head]`

Arguments:
- `hDC` - picture device context handle returned by the `#.GIFCreateDC` function.
- `rgb` - three element integer vector, or empty vector: transparent color.
- `chan` - integer scalar: channel number.
- `head` - optional simple character vector: additional HTTP headers.

Result: None.

Description:

The `DRAW_PNGA` function converts a picture drawn on the picture device context `hDC` into an HTTP response that contains that picture in PNG format and sends this response to the client, identified by the channel number `chan`. The `rgb` parameter is a three-element integer vector. Each element must be between 0 and 255 and defines intensity of red, green, or blue color. All together the values of `rgb` specify the transparent color of the resulting PNG image. If `rgb` is empty, the image will not be transparent.

The optional `header` argument specifies additional HTTP headers to be included in the result. If the `header` is not present, the function creates standard HTTP headers: "Content-Type", "Content-Length", "Date", "Last-Modified", and "Pragma: no-cache". The application may specify additional headers, if it needs to, like "Cookie". Each header must be terminated with CR-LF pair (`⎕AV[4 3]`).

The `DRAW_PNGA` function executes asynchronously in a separate Windows NT thread. It returns to the APL immediately, after the thread is started. The application must close the channel `chan` after the function returns. The `DRAW_PNGA` function destroys the picture device context `hDC`, after the response has been sent to the client. Thus, the application **must** de-select all GDI objects that were selected in this device context before it calls the function. Application **must not** use `hDC` handle after the call.

In all places, when an APL application does not need to re-use the picture device context, it should use `DRAW_PNGA` function instead of `DRAW_PNG` function. This will improve the overall server performance, especially on multiprocessor computers.

Example:

See example in the description of the `#.ISAP.DRAW_PNG` function.

5.14. #.ISAP.DRAW_PNGF

Syntax: `{size}←file #.ISAP.DRAW_PNGF hDC rgb`

Arguments:
- `hDC` - picture device context handle returned by the `#.GIFCreateDC` function.
- `rgb` - three element integer vector, or empty vector: transparent color.
- `file` - simple character vector: fully qualified output file name.

Result:
- `size` - simple one-element integer vector: the size of the created file in bytes.

Description:

The `DRAW_PNGF` function converts a picture drawn on the picture device context `hDC` into picture in PNG format and saves the picture in the specified `file`. The `rgb` parameter is a three-element integer vector. Each element must be between 0 and 255 and defines intensity of red, green, or blue color. All together the values of `rgb` specify the transparent color of the resulting PNG image. If `rgb` is empty, the image will not be transparent.

The `DRAW_PNGF` function executes synchronously. It does not return until the image file is created. It can be applied to the same picture device context `hDC` more than once, if needed.

5.15. #.ISAP.DRAW_XBM

Syntax: `image←{header} #.ISAP.DRAW_XBM hDC`

Arguments: `hDC` - picture device context handle returned by the `#.GIFCreateDC` function.

Result: `image` - HTTP response containing the x-xbitmap image.

Description:

The `DRAW_XBM` function converts a picture drawn on the picture device context `hDC` into an HTTP response that contains that picture in x-xbitmap format.

The optional `header` argument specifies additional HTTP headers to be included in the result. If the `header` is not present, the function creates standard HTTP headers: "Content-Type", "Content-Length", "Date", "Last-Modified", and "Pragma: no-cache". An application may specify additional headers, if it needs to, like "Cookie". Each header must be terminated with a CR-LF pair (`⎕AV[4 3]`).

The `DRAW_XBM` function can be applied to the same picture device context more than once. The result of the function should be sent to the client using the **WRITE** command with `'N'` translation mode.

Example:

See example in the description of the `#.DrawBitmap` function.

5.16. #.ISAP.GetBitmap

Syntax: `hBitmap size←#.ISAP.GetBitmap FileName`

Arguments: `FileName` - fully qualified bitmap file name.

Result: `hBitmap` - bitmap handle. This will be 0, if specified bitmap could not be loaded.
`size` - two-element integer vector: size of the bitmap in pixels (width, height).

Description:

The `GetBitmap` function loads a bitmap defined by the `FileName` argument and returns the bitmap handle, if successful. The `hBitmap` value can be used in other functions that manipulate bitmaps. The second item of the result is the width and the height of the bitmap in pixels.

Example:

See example in the description of the `#.DrawBitmap` function.

5.17. #.ISAP.GET_COOKIE

Syntax: *value←_ALL_HTTP #.ISAP.GET_COOKIE Name*

Arguments:

Name	-	HTTP cookie name.
_ALL_HTTP	-	environment variable created by the application manager.

Result: *value* - simple character vector, if cookie is defined, or an empty vector.

Description:

The *GET_COOKIE* function takes the value of the *_ALL_HTTP* environment variable, defined for the current channel by the application manager, and the cookie name as its arguments. It returns the value of the cookie with specified name, as a simple character vector. The result will be an empty vector if the cookie was not defined for the current HTTP request.

Example:

```
      _ALL_HTTP #.ISAP.GET_COOKIE 'LINGOID'
A1A8965A54
```

5.18. #.ISAP.GETHTML

Syntax: *res←chan #.ISAP.GETHTML FileName*

Arguments:

FIleName	-	file name relatively to the HtmlDir directory specified in application's initialization file (iserv.ini).
chan	-	integer scalar: the channel number.

Result: *res* - integer scalar: 1, if the function is successful, and 0 otherwise.

Description:

The *GETHTML* function loads the specified file in the channel's buffer. File name is defined relative to the HtmlDir directory that is defined in the application's initialization file. See section *Iserv.ini file settings*.

If the file was successfully loaded into the channel's buffer, the function returns 1. If the file could not be loaded the function executes the **SENDURL** command using the URL specified by the FileErrorURL setting in the application's initialization file and returns 0. The application must terminate the processing of the request if the result of the *GETHTML* function is 0.

The function should be used by the channel's program (in the thread latent expression, or in a function that supports an **EXEC** tag) before any output is sent to the client by **HEADER**, **WRITE**, **PARTITION**, **HTML**, or other similar command.

Example:

```
      _CHAN #.ISAP.GETHTML 'MYFILE.HTM'
1
```

5.19. #.ISAP.GETENTRY

Syntax: *value←#.ISAP.GETENTRY File Section Entry*

Arguments:

File	-	character vector: fully qualified name of a Windows initialization file.
Section	-	character vector: a section name in the initialization file.
Entry	-	character vector: an entry name in the section, or an empty vector.

Result:

value	-	nested vector of simple character vectors, simple character vector, or an empty vector.

Description:

The *GETENTRY* function returns the value of an *Entry* defined in a *Section* of a Windows initialization *File*. If the value of the *Entry* argument is an empty vector, the function returns all entry names for the specified section of the initialization file, as a nested vector of simple character vectors.

The result of the function will be an empty vector if the *File*, *Section*, or an *Entry* can not be found.

Example:

The following is the content of the C:\ISAP\ISERV\Iserv.ini initialization file:

```
[ISAP]
FileErrorURL=/isapdoc/ferror.htm
CommandErrorURL=/isapdoc/cerror.htm
AppErrorURL=/isapdoc/aerror.htm
HomeDir=c:\isap\iserv\
HtmlDir=c:\isap\Lingo\docs\
```

A few examples of the use of *GETENTRY* function:

```
      #.ISAP.GETENTRY 'C:\ISAP\ISERV\Iserv.ini' 'ISAP' ''
 FileErrorURL  CommandErrorURL  AppErrorURL  HomeDir  HtmlDir
      #.ISAP.GETENTRY 'C:\ISAP\ISERV\Iserv.ini' 'ISAP' 'HtmlDir'
c:\isap\Lingo\docs\
```

5.20. #.ISAP.IMAGE

Syntax: `#.ISAP.IMAGE hMetaFile (Width Height) rgb chan [head]`

Arguments:

`hMetaFile`	- the value of '*HANDLE*' property of Dyalog APL metafile object.
`Width`	- positive integer: horizontal size of the resulting image in pixels.
`Height`	- positive integer: vertical size of the resulting image in pixels.
`rgb`	- three-element integer vector, or an empty vector: transparent color.
`chan`	- simple integer scalar: channel number.
`head`	- optional simple character vector: additional HTTP headers.

Result: None.

Description:

The *IMAGE* function takes a metafile object's handle, creates the HTTP response containing the image in GIF format, and sends the response to the client, identified by the channel number `chan`. The `hMetaFile` argument should be a metafile handle that can be obtained as the metafile's *HANDLE* property. To avoid unwanted image transformation, the APL metafile object should use '*PIXEL*' coordinate system and the '*SIZE*' property set to the desired size of the resulting image.

The elements of the optional `rgb` vector must be integers between 0 and 255. Each element of the `rgb` vector defines intensity of red, green, and blue color. The resulting color specifies the transparent color of the GIF image. If the `rgb` vector is empty or omitted, the resulting image won't be transparent.

The `head` parameter specifies additional HTTP headers to be inserted in the HTTP response. Each header must be terminated with CR-LF pair (⎕*AV*[4 3]). If the headers argument is omitted or it is an empty vector, the function creates the following standard HTTP headers: `Date`, `Expires`, `Last-Modified`, `Content-Length`, `Content-Type`, `Pragma:no-cache`.

The *IMAGE* function executes asynchronously in a separate Windows NT thread. It returns to the APL immediately, after the thread is started. The application must close the channel `chan` after the function returns.

Example:

Creates memory based transparent GIF image from the *MF* metafile object and sends it to the client. Blue color in the image defined by the metafile will be transparent.

```
#.ISAP.IMAGE ('MF'⎕WG'HANDLE')(500 300)(0 0 255)_CHAN
```

5.21. #.ISAP.IMAGEF

Syntax: `len←file #.ISAP.IMAGEF hMetaFile (Width Height) rgb`

Arguments:

`hMetaFile`	- the value of `'HANDLE'` property of Dyalog APL metafile object.
`Width`	- positive integer: horizontal size of the resulting image in pixels.
`Height`	- positive integer: vertical size of the resulting image in pixels.
`rgb`	- three-element integer vector, or an empty vector: transparent color
`file`	- simple character vector: fully qualified file name.

Result: `len` - numeric scalar: the size of created image file.

Description:

The `IMAGEF` function takes a metafile object's handle and creates the file `file` containing the image in the GIF format. The `hMetaFile` argument should be a metafile handle that can be obtained as the metafile's `HANDLE` property. To avoid unwanted image transformation, the APL metafile object should use `'PIXEL'` coordinate system and the `'SIZE'` property set to the desired size of the resulting image.

The elements of the optional `rgb` vector must be integers between 0 and 255. Each element of the `rgb` vector defines intensity of red, green, and blue color. The resulting color specifies the transparent color of the GIF image. If the `rgb` vector is empty or omitted, the resulting image won't be transparent.

The `IMAGEF` function creates a GIF image on disk. The `file` parameter defines the output file name. The result is the number of bytes written to the file. If specified file already exists, it will be replaced. Of course, because the performance is important, you should avoid the creation of an intermediate file and build the complete HTTP response that contains the image in memory using the `#.ISAP.IMAGE` function.

Example:

Creates graphics file c:\image.gif from the *MF* metafile:

```
      len←'c:\image.gif'#.ISAP.IMAGEF('MF'⎕WG'HANDLE')(500 300)''
      len
8534
```

5.22. #.ISAP.IMAGEX

Syntax: `pic←{headers}#.ISAP.IMAGEX hMetaFile (Width Height)`

Arguments:

`hMetaFile`	- the value of `'HANDLE'` property of Dyalog APL metafile object.
`Width`	- positive integer: horizontal size of the resulting image in pixels.
`Height`	- positive integer: vertical size of the resulting image in pixels.
`headers`	- simple character vector: optional additional HTTP headers, or an empty vector.

Result: `pic` - simple character vector: the HTTP response.

Description:

The *IMAGEX* function takes a metafile object's handle and creates an HTTP response containing the image in the x-xbitmap format. The `hMetaFile` argument should be a metafile handle that can be obtained as the metafile's *HANDLE* property. To avoid unwanted image transformation, the APL metafile object should use `'PIXEL'` coordinate system and the `'SIZE'` property set to the desired size of the resulting image.

The `headers` parameter specifies additional HTTP headers to be inserted in the HTTP response. Each header must be terminated with a CR-LF pair (`⎕AV[4 3]`). If the headers argument is omitted or it is an empty vector, the function creates the following standard HTTP headers: `Date`, `Expires`, `Last-Modified`, `Content-Length`, `Content-Type`, `Pragma:no-cache`.

The result of the function is a simple character vector that should be sent to the client using the **WRITE** command with translation mode `'N'`.

Example:

Creates memory based x-xbitmap image from the *MF* metafile object and sends it to the client.

```
img←#.ISAP.IMAGEX ('MF'⎕WG'HANDLE')(500 300)
#.SAY 'WRITE' _CHAN 'N' img
```

5.23. #.ISAP.IMAGEXF

Syntax: *len←#.ISAP.IMAGEXF hMetaFile (Width Height) FileName*

Arguments:

hMetaFile	- the value of '*HANDLE*' property of Dyalog APL metafile object.
Width	- positive integer: horizontal size of the resulting image in pixels.
Height	- positive integer: vertical size of the resulting image in pixels.
FileName	- simple character vector: fully qualified file name.

Result: *len* - numeric scalar: the size of created image file.

Description:

The *IMAGEXF* function takes a metafile object's handle and creates the file *FileName* containing the image in the x-xbitmap format. The *hMetaFile* argument should be a metafile handle that can be obtained as the metafile's *HANDLE* property. To avoid unwanted image transformation, the APL metafile object should use '*PIXEL*' coordinate system and the '*SIZE*' property set to the desired size of the resulting image.

The *IMAGEXF* function creates an x-xbitmap image on disk. The *FileName* parameter defines the output file name. The result is the number of bytes written to the file. If specified file already exists, it will be replaced. Because performance is important, you should avoid the creation of an intermediate file and build the complete HTTP response that contains the image in memory using the *#.ISAP.IMAGEX* function.

Example:

Creates graphics file c:\image.xbm from the *MF* metafile:

```
      len←#.ISAP.IMAGEXF ('MF'⎕WG'HANDLE')(500 300)'c:\image.xbm'
      len
118534
```

5.24. #.ISAP.PIMAGE

Syntax: `#.ISAP.PIMAGE hMetaFile (Width Height) rgb chan [head]`

Arguments:

`hMetaFile`	- the value of `'HANDLE'` property of Dyalog APL metafile object.
`Width`	- positive integer: horizontal size of the resulting image in pixels.
`Height`	- positive integer: vertical size of the resulting image in pixels.
`rgb`	- three-element integer vector, or an empty vector: transparent color.
`chan`	- simple integer scalar: channel number.
`head`	- optional simple character vector: additional HTTP headers.

Result: None.

Description:

The `PIMAGE` function takes a metafile object's handle, creates the HTTP response containing the image in PNG format, and sends the response to the client, identified by the channel number `chan`. The `hMetaFile` argument should be a metafile handle that can be obtained as the metafile's `HANDLE` property. To avoid unwanted image transformation, the APL metafile object should use `'PIXEL'` coordinate system and the `'SIZE'` property set to the desired size of the resulting image.

The elements of the optional `rgb` vector must be integers between 0 and 255. Each element of the `rgb` vector defines intensity of red, green, and blue color. The resulting color specifies the transparent color of the GIF image. If the `rgb` vector is empty or omitted, the resulting image won't be transparent.

The `head` parameter specifies additional HTTP headers to be inserted in the HTTP response. Each header must be terminated with CR-LF pair (`⎕AV[4 3]`). If the headers argument is omitted or it is an empty vector, the function creates the following standard HTTP headers: `Date`, `Expires`, `Last-Modified`, `Content-Length`, `Content-Type`, `Pragma:no-cache`.

The `PIMAGE` function executes asynchronously in a separate Windows NT thread. It returns to the APL immediately, after the thread is started. The application must close the channel `chan` after the function returns.

Example:

Creates memory based transparent PNG image from the `MF` metafile object and sends it to the client. Blue color in the image defined by the metafile will be transparent.

```
#.ISAP.PIMAGE ('MF'⎕WG'HANDLE')(500 300)(0 0 255)_CHAN
```

5.25. #.ISAP.PIMAGEF

Syntax: `len←file #.ISAP.PIMAGEF hMetaFile (Width Height) rgb`

Arguments:

`hMetaFile`	- the value of `'HANDLE'` property of Dyalog APL metafile object.
`Width`	- positive integer: horizontal size of the resulting image in pixels.
`Height`	- positive integer: vertical size of the resulting image in pixels.
`rgb`	- three-element integer vector, or an empty vector: transparent color
`file`	- simple character vector: fully qualified file name.

Result: `len` - numeric scalar: the size of created image file.

Description:

The `PIMAGEF` function takes a metafile object's handle and creates the file `file` containing the image in the PNG format. The `hMetaFile` argument should be a metafile handle that can be obtained as the metafile's `HANDLE` property. To avoid unwanted image transformation, the APL metafile object should use `'PIXEL'` coordinate system and the `'SIZE'` property set to the desired size of the resulting image.

The elements of the optional `rgb` vector must be integers between 0 and 255. Each element of the `rgb` vector defines intensity of red, green, and blue color. The resulting color specifies the transparent color of the GIF image. If the `rgb` vector is empty or omitted, the resulting image won't be transparent.

The `PIMAGEF` function creates a PNG image on disk. The `file` parameter defines the output file name. The result is the number of bytes written to the file. If specified file already exists, it will be replaced. Of course, because the performance is important, you should avoid the creation of an intermediate file and build the complete HTTP response that contains the image in memory using the `#.ISAP.PIMAGE` function.

Example:

Creates graphics file c:\image.png from the `MF` metafile:

```
      len←'c:\image.png'#.ISAP.PIMAGEF('MF'⎕WG'HANDLE')(500 300)''
      len
8534
```

5.26. #.ISAP.PUTENTRY

Syntax: *res←#.ISAP.PUTENTRY File Section Entry Value*

Arguments:

File	-	character vector: fully qualified name of a Windows initialization file.
Section	-	character vector: a section name in the initialization file.
Entry	-	character vector: an entry name in the section.
Value	-	character vector: entry value, or an empty vector.

Result: *res* - numeric scalar: 1, if function is successful, or 0, otherwise.

Description:

The *PUTENTRY* function records a value in the specified initialization file under the specified section. The value is recorded as an `Entry=Value` string. The result of the function will be 1 if the function is successful.

Example:

The following is the content of the C:\ISERV\Iserv.ini initialization file:

```
[ISAP]
FileErrorURL=/isapdoc/ferror.htm
CommandErrorURL=/isapdoc/cerror.htm
AppErrorURL=/isapdoc/aerror.htm
HtmlDir=c:\isap\Lingo\docs\
```

After executing the following expression:

```
#.ISAP.PUTENTRY 'C:\ISERV\Iserv.ini' 'ISAP' 'Test' 'MyValue'
```

The new content of the file will be:

```
[ISAP]
FileErrorURL=/isapdoc/ferror.htm
CommandErrorURL=/isapdoc/cerror.htm
AppErrorURL=/isapdoc/aerror.htm
HtmlDir=c:\isap\Lingo\docs\
Test=MyValue
```

5.27. #.ISAP.repl

Syntax: *res←#.ISAP.repl String Pat Rep*

Arguments:

String	-	simple character vector or an empty vector.
Pat	-	simple character vector: search string.
Rep	-	simple character vector, or an empty vector: replacement string.

Result: *res* - simple character vector: result of replacement *Pat* with *Rep*.

Description:

The *repl* function effectively replaces all occurrences of *Pat* with *Rep* in the vector *String*.

Example:

```
      #.ISAP.repl 'aaacaa' 'c' ''
aaaaa
      #.ISAP.repl 'aaaaa' 'a' 'bb'
bbbbbbbbbb
```

5.28. #.ISAP.RUNPROGR

Syntax: `Msg←#.ISAP.RUNPROGR CmdLine`

Arguments: `CmdLine` - simple character vector: command line.

Result: `Msg` - simple character vector, or an empty vector.

Description:

The `RUNPROGR` function starts an external process using the `CmdLine` argument, as a command line. The function does not return until the process completes its initialization. The result is an empty vector, if the function was successful. The result is an error message if an error occurs.

Example:

```
#.ISAP.RUNPROGR 'notepad.exe c:\myfile.txt'
```

5.29. #.ISAP.TIME

Syntax: `time←#.ISAP.TIME offset`

Arguments: `offset` - integer scalar: offset from the current time in seconds.

Result: `time` - simple character vector: formatted HTTP time.

Description:

The `TIME` function returns the time relative to the current time in the HTTP standard. The `offset` argument specifies an offset from the current time. It can be positive (the future) or negative (the past).

Example:

```
      #.ISAP.TIME 0
Wed, 02 Dec 1998 21:18:20 GMT
      #.ISAP.TIME 3600
Wed, 02 Dec 1998 22:18:21 GMT
```

5.30. #.ISAP.TOLOWER

Syntax: `res←#.ISAP.TOLOWER String`

Arguments: `String` - simple character vector or an empty vector.

Result: `res` - simple character vector: the `String` with all letters converted to lower case.

Description:

The `TOLOWER` function effectively converts its argument to lower case.

Example:

```
      A←#.ISAP.TOLOWER 'Just Example 123'
      A
just example 123
```

5.31. #.ISAP.TOUPPER

Syntax: *res←#.ISAP.TOUPPER String*

Arguments: *String* - simple character vector or an empty vector.

Result: *res* - simple character vector: the *String* with all letters converted to upper case.

Description:

The *TOUPPER* function effectively converts its argument to upper case.

Example:

```
      A←#.ISAP.TOUPPER 'Just Example 123'
      A
JUST EXAMPLE 123
```

5.32. #.ISAP.TRBLANKS

Syntax: *res←#.ISAP.TRBLANKS String*

Arguments: *String* - simple character vector or an empty vector.

Result: *res* - simple character vector: the *String* with leading and trailing blanks removed.

Description:

The *TRBLANKS* function effectively removes all leading and trailing blanks from its argument.

Example:

```
      A←#.ISAP.TRBLANKS '  aaacaa   '
      A
aaacaa
      ⍴A
6
```

5.33. #.ISAP.DEB

Syntax: *res←#.ISAP.DEB String*

Arguments: *String* - simple character vector or an empty vector.

Result: *res* - simple character vector: the *String* with leading, trailing and embedded blanks removed.

Description:

The *DEB* function effectively removes all excessive blanks from its argument.

Example:

```
      A←#.ISAP.TRBLANKS '  a   caa  b '
      A
a caa b
      ⍴A
7
```

5.34. #.ISAP._PARAMETER

Syntax: `value←_IN #.ISAP._PARAMETER Name`

Arguments:

`Name`	-	simple character vector: CGI parameter name.
`_IN`	-	`_IN` environment variable for the channel.

Result: `value` - nested vector of character vectors, simple character vector, or an empty vector.

Description:

The `_PARAMETER` function retrieves the value of a CGI variable `Name`, passed with the HTTP request to the current channel. All CGI variables are kept in the `_IN` environment variable created by the application manager, when the channel's thread is initialized.

The result will be an empty vector, if the specified variable was not define. The result will be a simple character vector, if the specified variable was defined once. The result will be nested vector of simple character vectors, if the specified variable was defined more than once.

CGI variables are passed with the HTTP request when a CGI form generates the request. All variables defined for this form will be parsed by the application manager and kept in the `_IN` variable, which is global for the application thread. Some variables may have more than one value. For example, if the form contains a list with multiple selection allowed, the variable that represents this control will have more than one value if a user selects more than one option from the list.

Example:

```
      _IN #.ISAP._PARAMETER 'creditcard'
VISA
      _IN #.ISAP._PARAMETER 'selitems'
 cat  dog  rat
```

5.35. #.ΔOUT

Syntax: `#.ΔOUT String`

Arguments: `String` - nested vector of simple character vectors or a simple character vector.

Result: None.

Description:

The `ΔOUT` function logs a record in the application's log file. The log file tie number is a global variable `LOG_TN`. The log file is created by the `#.ISAP.CreateLogFile` function on system startup. The following code can be found in the `SETGLOBALS` function:

```
⍝ GET NAME OF THE LOG FILE
 LOG_TN←0
 →(0=⍴A←#.ISAP.GETENTRY FILE'ISAP' 'LOGFILE')↑L1
 LOG_TN←#.ISAP.CreateLogFile A
L1: . . . . . .
```

An application log always should be used. Otherwise it won't be possible to find errors in the production application running as a service. The application manager always logs all errors, if the log file was successfully created at start up. Applications may log their own specific information to the log file using the `#.ΔOUT` function. The function appends the current timestamp to any message submitted for logging.

Example:

```
      #.ΔOUT 'Execution started'
```

5.36. #.ISAP.DELCOOKIE

Syntax: *header←#.ISAP.DELCOOKIE Name*

Arguments: *Name* - simple character vector: cookie name to delete.

Result: *header* - simple character vector: HTTP header.

Description:

The *#.ISAP.DELCOOKIE* returns properly formed HTTP header that should be sent to the client in order to delete cookie *Name* from client's computer. The result of the function can be used in the **HEADER** or **HTML** commands. It also can be used in the APL expression in the **FINISH** tag. The function deletes only cookie set for the path defined by the *_PATH* thread environment variable and all paths below it.

Example:

The following FINISH tag will send to the client the resulting HTML document and will delete cookie that has name VISIT from the client computer:

```
<--#ISAP FINISH="#.ISAP.DELCOOKIE 'VISIT'"-->
```

5.37. #.ISAP.SETCOOKIE

Syntax: *header←Name #.ISAP.SETCOOKIE Value*

Arguments: *Name* - simple character vector: cookie name to set.
Value - simple character vector: cookie value to set.

Result: *header* - simple character vector: HTTP header.

Description:

The *#.ISAP.SETCOOKIE* returns properly formed HTTP header that should be sent to the client in order to set value *Value* for the cookie *Name* on client's computer. The result of the function can be used in the **HEADER** or **HTML** commands. It also can be used in the APL expression in the **FINISH** tag. The function sets only cookie for the path defined by the *_PATH* thread environment variable and all paths below it. The cookie set will not have the expiration date. Thus, it will be deleted, when user closes the browser.

Example:

The following FINISH tag will send to the client the resulting HTML document and will set value 5 to the cookie that has name VISIT on the client computer:

```
<--#ISAP FINISH="'VISIT' #.ISAP.SETCOOKIE '5'"-->
```

5.38. _PARSE

Syntax: `{rc}←_PARSE URL`

Arguments: `URL` - simple character vector: URL that defines a new script file.

Result: `rc` - simple numeric scalar `0` or `¯1`.

Description:

The **_PARSE** function loads a new script file in the channel's buffer. The script file is defined by the `URL` argument. The `URL` must address a file on the current server and, therefore, should not contain the protocol definition and server name. That means it always should start with "/" character.

The function parses the script file and creates a new thread program. The function copies the content of the namespace defined in the **WS** tag of the new script file into the current thread (if the **WS** tag is specified and it differs from the current namespace). If file loaded does not contain any executable elements, the file is sent to the client by the **HTML** command. If file could not be found, the function redirects client to the URL defined under `FileErrorURL` entry in the ISERV.INI file. If any error occurs during the thread program setup (for example, when application namespace is missing), the function redirects client to the URL defined under `AppErrorURL` entry in the ISERV.INI file.

The function returns `0`, when the thread program must be terminated. This happens when **_PARSE** redirects client to a different URL because of error, or when loaded file does not contain executable elements and was sent to the client.

The function returns `¯1`, when the script file was successfully loaded and a new thread program was compiled.

When **_PARSE** is used inside the defined function that implements an **EXEC** tag, the tag's function must return to the application manager the result returned by the **_PARSE** function.

When **_PARSE** is used inside thread latent expression the application should analyze the value of `rc` returned by the function. The latent expression exits, if `rc` is `¯1`. The latent expression must set the value of the `_COM` global variable to an empty vector and exit, if `rc` is `0`.

Example:

See example in the description of the **PARSE** command.

5.39. checked

Syntax: `rc←checked`

Arguments: None.

Result: `rc` - simple character vector: `' checked'`.

Description:

Defines a useful constant to be used in **EXEC** tags.

Example:

See example in the description of the **if** function.

5.40. selected

Syntax: *rc←selected*

Arguments: None.

Result: *rc* - simple character vector: ' *selected*'.

Description:

Defines a useful constant to be used in **EXEC** tags.

Example:

See example in the description of the **if** function.

5.41. if

Syntax: *rc←val if cond*

Arguments: *val* - simple character vector.
cond - Boolean scalar.

Result: *rc* - is *val*, if *cond* is 1, and an empty vector otherwise.

Description:

This function could be used in **EXEC** tags for simple manipulations with elements of HTML.

Example:

The following fragment of a script file, uses **if** function to restore the selection of radio button:

```
. . . . .
<!--#ISAP NVAR=fnc NAME="fnc" DEFAULT="0"-->
. . . . .
Select function to execute:<br>
<input type=radio name="fnc" value="0"<!--#ISAP EXEC="selected if fnc=0"-->>
- First<br>
<input type=radio name="fnc" value="1"<!--#ISAP EXEC="selected if fnc=1"-->>
- Second<br>
<input type=radio name="fnc" value="2"<!--#ISAP EXEC="selected if fnc=2"-->>
- Third<br>
```

5.42. in

Syntax: *rc←val in set*

Arguments:

val	-	simple character vector or numeric scalar.
set	-	simple numeric vector or nested vector of character vectors.

Result: *rc* - is 1, if *val* can be found in *set*, and 0 otherwise.

Description:

This function could be used in **EXEC** tags for simple manipulations with elements of HTML.

Example:

The following fragment of a script file, uses **if** and **in** functions to restore the selection of elements of list box:

```
. . . . .
<!--#ISAP NVAR=menu NAME="menu" DEFAULT="0"-->
. . . . .
Select products from menu:<br>
<select name="menu" multiple size="5">
<option value="0"<!--#ISAP EXEC="selected if 0 in menu"-->>Option 1
<option value="1"<!--#ISAP EXEC="selected if 1 in menu"-->>Option 2
<option value="2"<!--#ISAP EXEC="selected if 2 in menu"-->>Option 3
<option value="3"<!--#ISAP EXEC="selected if 3 in menu"-->>Option 4
<option value="4"<!--#ISAP EXEC="selected if 4 in menu"-->>Option 5
</select>
. . . .
```

5.43. takes

Syntax: *rc←name takes val*

Arguments:

name	-	simple character vector.
val	-	arbitrary array.

Result: *rc* - is *val*.

Description:

This function assigns value *val* to variable, which name is defined by *name*. This function could be used in **EXEC** tags for simple manipulations with elements of HTML.

Example:

The following fragment of a script file, uses **takes** function to increment value of CGI variable:

```
. . . . .
<!--#ISAP NVAR=cnt NAME="cnt" DEFAULT="0"-->
. . . . .
Current value of cnt is <!--#ISAP EXEC="cnt"-->.
<form method=post action="sample.xml">
<input type=hidden name="cnt" value="<!--#ISAP EXEC="'cnt' takes cnt+1"-->">
<input type=submit value="Next">
</form>
. . . .
```

5.44. is

Syntax: *rc*←*val1 is val2*

Arguments: *val1*, *val2* - arbitrary arrays.

Result: *rc* - is 1, if *val1* is equivalent to *val2*; it is 0 otherwise.

Description:

This function checks whether *val1* and *val2* are the same. This function could be used in **EXEC** tags for simple manipulations with elements of HTML.

Example:

The following fragment of a script file, uses **if** and **is** functions to restore the selection of an element in the selection box:

```
. . . . .
<!--#ISAP CVAR=menu NAME="menu" DEFAULT="Menu 1"-->
. . . . .
Select products from menu:<br>
<select name="menu" size="5">
<option value="Menu 1"<!--#ISAP EXEC="selected if menu is 'Menu 1'"-->>Prod 1
<option value="Menu 2"<!--#ISAP EXEC="selected if menu is 'Menu 2'"-->>Prod 2
<option value="Menu 3"<!--#ISAP EXEC="selected if menu is 'Menu 3'"-->>Prod 3
<option value="Menu 4"<!--#ISAP EXEC="selected if menu is 'Menu 4'"-->>Prod 4
<option value="Menu 5"<!--#ISAP EXEC="selected if menu is 'Menu 5'"-->>Prod 5
</select>
. . . .
```

6. Isap.ini File Settings

The ISAP.INI file must be in the same directory where the ISAP.DLL file is located. This file controls how the Internet Server Auxiliary Processor runs. Failure to set entries in the ISAP.INI file correctly will prevent ISAP from starting and/or running correctly.

ISAP is loaded into the Internet Information Server's process space when the very first HTTP request that refers to ISAP.DLL explicitly or implicitly (request refers to file with the extension ".xml", or with other registered extensions) arrives. ISAP attempts to start Dyalog APL for Windows using the information in the ISAP.INI file.

There are two sections in this file. The `[StartUp]` section defines ISAP's execution mode and controls settings for the production mode (see below). The [Development] section is used when the development version of the APL interpreter is used. Production applications do not need to use this section at all.

A typical ISAP.INI file is shown below:

```
[StartUp]
WS=C:\DYALOG95\ISERV\ISERV
maxws=10000
ErrorURL=/doc/error.htm
DEBUG=1
DYALOG=c:\runtime

[Development]
DYALOG=C:\DYALOG95
maxws=20000
session_rows=16
session_cols=77
edit_rows=15
edit_cols=70
sm_rows=25
sm_cols=79
trace_offset_x=0
trace_offset_y=0
wspath=C:\DYALOG95\
aplnid=0
lines_on_functions=1
aplk=unify_us.din
aplt=win
aplkeys=C:\DYALOG95\aplkeys
apltrans=C:\DYALOG95\apltrans
autotrace=1
fontsize=16
monitor=0
term=win
apl_font=apl
quadcmd=C:\DYALOG95
session_file=C:\DYALOG95\def_us.dse
edit_first_x=0
edit_first_y=0
edit_offset_x=3
edit_offset_y=3
trace_first_x=0
trace_first_y=0
```

Entries in the `[StartUp]` **section:**

`WS` Fully qualified path to the workspace file to be loaded when ISAP starts. The path ***must not*** contain the file extension. The extension ".dws" is assumed. The specified workspace file will be used when either the production or development version of the APL is run.

`maxws` Specifies the workspace size in kilobytes, when ISAP is running in production mode.

`ErrorURL` Specifies the URL that should be sent to the client when ISAP cannot communicate with Dyalog APL. If APL unexpectedly exits, ISAP remains in the memory and running until the WWW service is stopped. Obviously, no useful job can be performed. If ISAP detects such a situation, it will respond with the URL specified in `ErrorURL` entry. This URL must satisfy the conditions described in the **SENDURL** command. This entry can be omitted. If so, the ISAP responds with the standard message: *Service unavailable.*

`DEBUG` If this entry is present, it specifies the ISAP execution mode: 1 - debugging mode, 0 - production mode. If the entry is omitted, the production mode is assumed. In production mode ISAP will start the dyalogrt.exe file that must be in the directory specified by the `DYALOG` parameter (see below). In production mode the APL interpreter runs as a Windows NT service. If development mode is specified, ISAP will start the dyalog.exe file using entries in the `[Development]` section of the ISAP.INI file. The application should analyze the value of the *`APLVERSION`* property of the Dyalog APL root object in order to detect what version (development or production) of the software is running. See important notes about development and production execution modes in the Section 7.

`DYALOG` If this entry present, it specifies the directory where APL interpreter's run-time executables are located. If this entry is not present, dyalogrt.exe file must be in the same directory where the ISAP.DLL file resides. Normally, the production application does not need to use this parameter. It should keep all executable files in the same directory. However, when running more than one copy of ISAP and APL, this parameter is useful. All copies of the ISAP.INI files for different copies of the ISAP.DLL files should point to the same directory to increase the performance of the server. See section *Running Multiple Copies of the ISAP* for more info.

Entries in the `[Development]` **section:**

This section must properly describe the desired Dyalog APL configuration if you want to develop Internet applications on this installation of the Internet Server. This section may not be present on computers that run the production version of ISAP only. ISAP will create a registry folder:

HKEY_LOCAL_MACHINE\Software\Lingo\ISAP

This folder will keep the specified parameters from the `[Development]` section. When ISAP starts the development version of Dyalog APL (file dyalog.exe), it passes the created registry folder to APL as a startup initialization file. Thus, any entry from the `[Development]` section will be passed to Dyalog APL as a startup parameter. You **must** define at least the following entries in this section:

`DYALOG` Points to the root installation directory of Dyalog APL (the directory, where dyalog.exe file is located).

`maxws` Specifies the workspace size in kilobytes.

ISAP will use the `WS` and the `ErrorURL` entries from the `[StartUp]` section when starting the development APL system. See Dyadic's documentation regarding other settings in the `[Development]` section.

7. Developing ISAP Applications

This chapter describes programming conventions for the ISAP application manager supplied with the product. Users may develop their own application managers, if they wish. Lingo Allegro USA, Inc. will support only the application manager supplied with the product.

The ISAP application manager is the Dyalog APL workspace found at c:\isap\iserv\iserv.dws. The *ISERV* workspace contains the application manager itself and a few demo applications written for ISAP. You may delete all objects from the *ISERV* workspace except the objects located in the root namespace and the *ISAP* namespace.

7.1. Iserv.ini File Settings

The ISERV.INI file is used by the *ISERV* application as its initialization file. It keeps global definitions and simplifies the installation and maintenance of the application. The application expects this file in the same directory where the workspace itself resides. There is only one section in this file - `[ISAP]`. It contains the following entries:

`HomeDir` Specifies the root directory (terminating with "\" is required) for the directories where application files are stored. In our case this is the same directory where the workspace resides. It was set this way to simplify the installation only. You might want to use a different directory or to use more than one directory relatively to the root directory. These can be template HTML files, bitmap files, database files, etc.

`HtmlDir` Specifies the location of the HTML files on the server. This entry is used by the *#.ISAP.GETHTML* function, for example. Applications should avoid using physical references to the HTML files. Applications should use the **MAPURL** command to map virtual paths to the physical paths instead.

`FileErrorURL` Defines the URL for the document to be sent to the client when an application template file is not found. This is used only by the *#.ISAP.GETHTML* function.

`CommandErrorURL` Defines the URL for the document to be sent to the client when the specified application is not found (see next section).

`AppErrorURL` Defines the URL for the document to be sent to the client when an unhandled application error occurs (see next section).

`LogFile` Defines a fully qualified log file name. If the file does not exist, it will be created.

You may add new sections and/or entries to this file to satisfy your own needs. When you install the product for the first time, the installation program makes all necessary changes to that file. You should modify the *SETGLOBALS* function in order to grab more initialization parameters, if needed. Use the *#.ISAP.GETENTRY* function to read entries from the initialization file. The application manager executes the *SETGLOBALS* function when the APL workspace is loaded.

The application should analyze the value of the *APLVERSION* property of the APL root object to determine its execution mode. When the application runs in the debugging mode (defined by ISAP.INI), it can display some messages in the session window and interact with the user by displaying windows and dialog boxes. One important thing is that the application must not execute ⎕*OFF* on system shutdown, instead it must simply exit the main function. The application manager creates a global variable *DEBUG* in the root namespace that shows the version of the interpreter running.

In production mode, when the `DEBUG` flag in ISAP.INI file is not set, ISAP works as a Windows NT service. In this mode the APL application **absolutely must not** interact with the user, display modal windows and dialog boxes. The APL application must not exit into immediate execution mode either. If these conditions are not met, the APL application will be blocked. You will have to restart the NT server in order to run them again. They only way for the production-mode application to trace errors and non-standard situations is through the use of log files. The application manager must use the ⎕*OFF* function when ISAP exits.

7.2. Application Manager Start Up

When ISAP starts, the APL interpreter loads the workspace specified in the ISAP.INI file. The workspace must contain the latent expression that starts the application manager. The application manager supplied with the product shown below:

```
      ∇ ISERV;DEBUG;_SAVE;⎕TRAP
[1]     SETGLOBALS
[2]     ∆OUT'Loading...'
[3]     →SHARE↓EXIT
[4]     ∆OUT'Ready'
[5]     ⎕DQ'.'
[6]   EXIT:END
      ∇
```

The application manager performs its initialization by executing the *SETGLOBALS* function. This function should read the application initialization file and set global variables, like paths, URLs, execution mode, and so on. The application manager should not perform any heavy operations at this stage, like connecting to databases or reading network files. Any operations of such kind should be left to particular applications.

Next, the application manager executes the *SHARE* function that establishes connection with ISAP, attaches external functions from DLLs, create the invisible notification window (has name *_ISF*), and issues the **NOTIFY** command to ISAP. The application manager has not more than 30 seconds to issue the **NOTIFY** command. The Internet server will shutdown the APL interpreter if the command is not received within 30 seconds.

The application manager maintains two execution modes: development and production. In the development mode only errors 667 that are generated by the *SAY* function are trapped. In the production mode all errors are trapped. If an error occurs, the application manager will terminate the application thread and will close the corresponding ISAP channel.

7.3. ISAP Applications and Threads

An ISAP application is a namespace that contains everything that is needed to run the application. The functions in the application namespace can refer to functions in other namespaces using namespace qualifiers relatively to the root namespace only. The application namespace does not run itself. Instead, the application manager creates a copy of the application namespace called *request namespace* every time the application needs to be executed. Normally, the request namespace exists only during the processing of a single HTTP request. For this reason, when the application needs to store some global data to be used later for processing other HTTP requests, it should store those data in the original application namespace or in the root namespace, but not in the namespace that is currently executed.

An application may not define any own namespace. In this case its execution will start in an "empty" namespace, which contains only ISAP standard functions, environment variables, and CGI variables. The Application Manager keeps all standard functions, which are copied to any request namespace in the *ISAPCLI* namespace. Programmers may not delete any functions from this namespace, but they can add their own functions and variables, which will be copied in each request namespace before it starts the execution.

When the application manager is invoked, as the result of the arrival of the ISAP notification message, it creates a new Dyalog APL *execution thread* based on the *ISAPTHREAD* function. All execution threads are executed simultaneously by the APL interpreter. Applications should control the access to common critical resources using the *:Hold* control structure, if needed. Except this, applications should behave, as no other threads are running at the same time.

If an application needs to perform a specific initialization, like to connect to a database, or read a file, it should use global variables in its namespace that show the current status of the application. These variables will be copied in every request namespace when a new execution thread is created. For example, if an application located in the namespace *SURVEY* needs to read a survey file in order to run it could use the following technique:

```
      ∇ SurveyInit
[1]     :Hold 'survey'
[2]        :If 0=⎕NC'DATA'
[3]           #.SURVEY.DATA←DATA←ReadSurveyFile
[4]        :EndIf
[5]     :EndHold
      ∇
```

The function *SurveyInit* must be executed every time an application thread starts executing. If the global variable *DATA* exists, it means that the application's initialization has been performed already. *SurveyInit* simply exits in such case. If *DATA* is not defined, the application reads the file using the *ReadSurveyFile* function. The application stores the result of the function in its own application thread that is currently executing and in the original application namespace. This will guarantee that further threads of the same application will not have to read the file again.

7.4. Application Thread Initialization

ISAP sends a notification message to the application manager when one of the following occurs:

- The user has requested a document that has the extension ".xml". You may register other extensions to identify files that will be processed by ISAP.DLL. For example, the application manager will be called when a user clicks the OK button in the browser window that displays the following document:

```
<HTML><BODY>
<!--#ISAP WS=TEST-->
<P>
Click the button:
<P><CENTER>
<FORM METHOD="POST" ACTION="/ISAPDOC/TEST.XML">
<INPUT TYPE=SUBMIT VALUE="OK">
</FORM>
</BODY></HTML>
```

- The user referred to ISAP.DLL explicitly. For example, the processing of the following file will call ISAP to produce the image:

```
<HTML><BODY>
<P>
ISAP-generated image:
<P><CENTER>
<IMG SRC="/ISAP/ISAP.DLL?WS=COUNTER&VAL=15">
</BODY></HTML>
```

When the application manager receives the notification message from the ISAP, it performs the following actions:

1. A new APL thread, based on *ISAPTHREAD* function is started.
2. The application manager executes the **GETINFO** command on the channel that posted the message.
3. The application manager processes the ISAP standard tags.
4. The application manager finds the application to be executed. If an ".xml" file is requested, the application manager tries to locate the **WS** tag that identifies the application name (the namespace name). If the **WS** tag was not specified, the application manager is looking for a CGI variable WS that might be sent with the HTTP request to identify the application namespace. If the namespace was specified, the application manager creates a new namespace and copies the content of the application namespace into the newly created application thread. If the application name was not specified, an empty application thread is created.

5. The application manager copies the content of the *ISAPCLI* namespace into newly created thread namespace. This namespace contains functions, which are used internally by the application manager, and a few helper functions, which need run in the context of the thread namespace.
6. The application manager creates HTTP environment variables in the application thread.
7. The application manager defines CGI variables, for each of **CVAR** and **NVAR** tags that might be found in the referenced ".xml" file.
8. The application manager compiles the thread's program using **EXEC** tags defined in the referenced ".xml" file and thread's latent expression (see below).
9. The application manager executes the compiled program. If the thread's program is empty, the application manager returns the requested ".xml" file to the client without any processing. If the HTTP request refers to ISAP.DLL directly and the thread's program is empty, an error response is sent to the client.

7.5. Thread's Environment Variables

The application manager creates the following global variables when an application thread is created. All variable names start with "_". Programmers should not use names in their applications that start with this character. All variables can be referred directly within the current thread without namespace specification.

_CHAN	the IIS channel number that issued the HTTP request. The application should request data and send output to this channel only.
_PATH_TR	simple character vector: translated (physical) path to the requested document.
_FILE	numeric scalar: 1, if the script (template) file is loaded in the channel's buffer. 0 - otherwise. This variable shows whether the user request an ".xml" file or referenced ISAP.DLL directly.
_METHOD	simple character vector: the HTTP request method. Can be "GET" or "POST".
_QUERY	simple character vector: request's query string.
_PATH	simple character vector: virtual path to the requested document.
_CONTENT	simple character vector: the value of the HTTP Content-Type header.
_REMOTE_IP	simple character vector: client's IP address.
_REMOTE_HOST	simple character vector: the name of the remote host.
_REFFERER	simple character vector: URL of the referring page.
_USER	simple character vector: user name.
_UMAPPED_USER	simple character vector: unmapped user name.
_SERVER	simple character vector: server address.
_PORT	numeric scalar: server port number.
_SECURE	numeric scalar: 1, if this is a secure connection, 0 - otherwise.
_URL	simple character vector: base portion of URL.
_ACCEPT	simple character vector: all HTTP Accept headers.
_AUTH	simple character vector: authentication mode.
_ALL_HTTP	nested 2-column matrix: all HTTP headers. Each element of the matrix is a simple character vector. First column: header name. Second column: header value.
_BYTES	simple numeric scalar: number of pending bytes from the client. These data can be read by the **READ** command.
_IN	nested 2-column matrix: all CGI variables, posted with the request. Each element of the matrix is a simple character vector. First column: variable name. Second column: variable value.
_WS	Simple character vector: the original application's namespace name.
_COM	Nested vector of simple character vectors: thread program.

7.6. Thread's CGI Variables

The application manager creates global CGI variables for every **CVAR** and **NVAR** tag found in the script file, before the thread starts executing. The variables will have names specified in the appropriate tags, and will be global for this thread. Applications can refer to them directly, without namespace specification.

Applications can also maintain their own global variables to keep the execution history. The application manager creates a fresh copy of any new application thread using the original application namespace. Thus, if the application wants to store some values to use them in later runs, it should store those values in the original namespace, not in the current namespace that will be deleted when thread terminates.

7.7. Thread Program and Thread Latent Expression

The thread program is compiled by the application manager and consists of calls to APL functions that support **EXEC** tags found in the script file, and the thread latent expression.

The *thread latent expression* is a nested vector of simple character vectors. This vector must have the name *_COM*. Any element of this vector must be a valid argument for the "execute" primitive (⍎). The application manager will execute the thread latent expression step-by-step after all **EXEC** and **PART** tags were processed, if any. The thread latent expression must always be used when the application manager processes an HTTP request that refers to ISAP.DLL directly, because such a request cannot have an associated script file. Also, such a request must always have the CGI variable WS specified. Otherwise, the application manager won't be able to find the application to execute. See the *Clock* demo. Programmers should avoid the direct referencing the ISAP.DLL. The appropriate script files should be used instead. See the *Calculator* demo.

The thread latent expression also can be used when the use of **EXEC** tags is not convenient. Usually, this happens when an application should perform some actions that have side effects but do not produce any output to the script (template) file. Another case might be, when there is no a satisfactory way to describe an APL expression in an **EXEC** tag. In this case, the application may use a latent expression and replace custom tags explicitly using the **REPLACE** command.

The latent expression will usually be needed when an application outputs data "on-line", without storing the output in the template file in order to increase the performance. Programmers should avoid the use of thread's latent expression, when possible.

The APL interpreter executes all existing application threads simultaneously by implementing the time-sharing technique. Many ISAP commands and functions run as separate Windows NT threads. Thus, on multiprocessor systems the application manager can take advantage of using many physical processors in parallel, even when a single-threaded APL interpreter is used. The overall performance increase can also be achieved by using asynchronous auxiliary processors, which are run in separate Windows NT threads. For database access, Lingo Allegro's AP747 auxiliary processor is recommended.

If an application needs to pass data between application steps, it should use thread (namespace) global variables. These global variables exist only while the thread itself exists. For maintaining the "global history", the application should use the original application namespace to store variables that will remain in the workspace even after the application thread is destroyed.

7.8. Typical Steps Needed To Create an ISAP Application

First, design a set of HTML files that the application will use. Use **INCLUDE** tags, if the HTML files should contain often-used standard pieces. Plan forms and CGI variables that need to be passed between different documents. Use **CVAR** and **NVAR** tags to define those variables in the application thread. Locate places in the documents where the dynamic content should be placed and use **EXEC** tags to specify necessary actions that the application should perform to generate the document.

Next, you have to decide how many and what application namespaces you need to create. The simplest way is to have one namespace for each ".xml" document, and for each dynamically created image. This may mean that to process a single document you might need more than one namespace, because the browser may call ISAP more than once to generate the output document. Even if you keep all APL code in a single namespace, you should remember that request's namespace exists only while an HTTP request is processed.

Next step is to develop APL functions that support **EXEC** tags. If the document cannot be processed using **EXEC** tags only, write APL code to be used as a thread latent expression.

The following is an example of programming an ISAP application. This example is similar to the *Calculator* demo supplied with the application manager. The application displays a text control to accept arithmetic expressions from a user. It shows the result of the entered expression as a graphical image. If the entered expression cannot be evaluated, zero is returned as a result.

The application will use one namespace *CALC* and two script files `test.xml` and `display.xml`. The /isapdoc/test.xml, is shown below:

```
<!--#ISAP WS=CALC-->
<HTML><BODY>
<P><CENTER>
Type an expression and click "Execute":
<BR>
<!--#ISAP CVAR=EXPR NAME="EXPR" DEFAULT=""-->
<FORM ACTION="/ISAPDOC/TEST.XML" METHOD=POST>
<INPUT TYPE=TEXT NAME="EXPR" SIZE="30" VALUE="<!--#EXEC="EXPR"-->">
<BR>
<INPUT TYPE=SUBMIT VALUE="Execute">
</FORM>
<BR>
<IMG SRC="/ISAPDOC/DISPLAY.XML?VAL=<!--#EXEC="CALC EXPR"-->">
</BODY></HTML>
<!--#ISAP FINISH=""-->
```

This document defines one form that references the same document again, and an image. The form has a text input control named EXPR and a button without a name. When a user requests /isapdoc/test.xml from the Internet browser, ISAP.DLL is called. The application manager will find the **WS** tag in the document. It will create a new request namespace and will copy the content of the *CALC* namespace into that namespace.

Next, the application manager will process the **CVAR** tag that defines the *EXPR* thread CGI variable. When the document is loaded for the first time, this variable will not exist, so the application manager will create the *EXPR* variable that will be an empty character vector, as specified by the DEFAULT clause of the **CVAR** tag.

After that the application manager will compile the program for two **EXEC** tags and the thread latent expression and will start the execution of the thread program.

The first **EXEC** tag, which is used to display the previously entered expression into the text box, does not need any programming support. This tag will be replaced by the application manager with the value of the *EXPR* global variable that was defined by the **CVAR** tag.

The second **EXEC** tag, which is used to define the URL for the image, executes the APL function *CALC*. The function takes the value of *EXPR*, as its argument. The result of that function will be the result of the expression that user typed in the text box:

```
      ∇ VAL←CALC EXPR;VAL;⎕TRAP
[1]    ⍝ SET DEFAULT VALUE FOR THE RESULT. TRAP ERRORS
[2]     VAL←'0' ⋄ →(0=⍴EXPR)↑0 ⋄ ⎕TRAP←0 'E' '→0'
[3]    ⍝ EXECUTE EXPRESSION
[4]     VAL←⍎⍕EXPR
      ∇
```

After the *CALC* function exits, the URL of the <IMG> tag will be properly formed. It will look like:

<IMG SRC="/isapdoc/display.xml?VAL=*value*">

The execution of the thread ends with the FINISH tag that sends the result document to the user. The Internet browser will request file /isapdoc/display.xml in order to display image. This file is show below:

```
<!--#ISAP WS=CALC-->
<!--#ISAP CVAR=VAL NAME="VAL" DEFAULT="0"-->
<!--#ISAP EXEC="DISPLAY VAL"-->
```

This script instructs the application manager to make copy of the CALC namespace, define the value of VAL variable and to call the *DISPLAY* function. The function *CALC.DISPLAY* is shown below:

```
      ∇ R←DISPLAY VAL
[1]    ⍝ CONVERT TO NUMBER
[2]     VAL←⊃2⊃⎕VFI VAL
[3]    ⍝ CREATE PICTURE
[4]     VAL←#.GRAPHICS.LEDDISPLAY VAL
[5]    ⍝ SEND IMAGE
[6]     #.SAY'WRITE' _CHAN 'N' VAL
[7]     R←''
      ∇
```

The *SHOWIMAGE* function converts the variable into a number and calls the *LEDDISPLAY* library function in the *GRAPHICS* namespace to generate the output image. After that it sends the image back to the client.

7.9. Development Tips

- If you intend to use the ISERV application manager for your development (recommended), note that all new objects that are created after the application manager is started will be deleted when it is stops. So, you have to add new objects (namespaces, functions, or global variables) *before* you run the application manager. To do this,)*XLOAD* the workspace, create or copy all new objects you want, and)*SAVE* the workspace.
- Do not use the root namespace and the ISAP workspace to keep your own static functions and variables. It will be difficult to upgrade the ISAP versions. If you have functions and/or variables that are used by many application namespaces, create a new namespace (UTILS, LIB, etc.) and keep those common objects there, not in the root namespace. Use absolute namespace specification when referencing those objects: *#.UTIL.fmt*.
- Do not use names that starts with "_" character in your applications. They are reserved for the system use.
- During debugging, remember that you always get an error in the client namespace (application thread). In order to fix an error permanently you have to make changes in the original application namespace. Otherwise, this error will appear again when a new request is processed.
- Before starting debugging, stop all three Internet services (WWW, FTP, Gopher) using the Control Panel or the Internet Service Manager. After that, start the WWW service only. When debugging, WWW service should be the only Internet service running on the server.
- Before hitting the server with an HTTP request, save the workspace (if needed) and close the APL session that may be running as the result of a previous debugging session.

- When you have fixed an error, stop the WWW service. This will stop the application manager also. Save the workspace. Close Dyalog APL. Don't leave the session window open, this may prevent the WWW service to re-start. Start the WWW service again.
- Don't generate complete HTMLs in your applications. Use HTML templates instead (surrogate files that contain tags that need to be replaced with dynamically created content). Use files with extensions that are registered as ISAP script files (".xml" by default). If you need to change the template file during the processing of the request, use hidden fields in your HTML files to specify alternate template files that might be loaded by the *ISAP.GETHTML* or *_PARSE* functions. This will make your application more flexible.
- Use the ".xml" (or any other registered extension) as an extension for template file names. It will increase the performance of your applications, because you will not need to load template files explicitly. When the template file is loaded by ISAP implicitly, the ISAP processes standard tags in a separate Windows NT thread that increases the performance of the server.
- Always use log files to trace application behavior. Remember that in production mode you cannot use any windows to display error messages. Log files are the only possible way for you to know what is going on with your application.
- When developing your first application, you can take the supplied ISERV namespace, as a starting point. You may delete all objects from this workspace, except root, *ISAP*, and *ISAPCLI* namespaces.

8. Running Multiple Copies of the ISAP

The Internet Server Auxiliary Processor is a multithreaded application that tries to run many physical threads simultaneously. On servers that have more than one processor these threads will physically run in parallel. However, because the APL interpreter is a single-thread application, APL will be a limitation factor for the server's performance. In order to increase the performance of the Internet server, you can run more than one copy of ISAP and APL interpreter for different Internet applications.

You need to perform the following actions to run another copy of ISAP and APL:

1. Create a new directory for ISAP.DLL and ISAP.INI.
2. Make this directory available for the Internet Server using the Internet Server Manager, and assign an alias for this directory. Set EXECUTE access mode for the directory.
3. If you want to use script files with ISAP, register some file extension for the scripts and assign ISAP.DLL located in the created directory, as a script interpreter.
4. Point the `DYALOG` entry in the `[StartUp]` section of the ISAP.INI file to the same directory that the first copy of the ISAP is using.
5. Create a new directory for the script file and register it with the Internet Server. Specify both READ and EXECUTE access modes for this directory. The directory should contain regular HTML files and script files with the registered extension.
6. You may use the same or a different APL workspace that contains the application manager. If your applications have virtual paths and/or the script files extension hardcoded in the APL code, you must use a different workspace that reflects the changes.

Your applications running from the second application manager should be programmed as though there is only one copy of ISAP running, unless you want to share some data between your APL applications. Any number of instances of ISAP can be run on the same Windows NT server, if you follow the rules described above. These instances of ISAP and APL will be absolutely independent and will run simultaneously in Windows.

9. Registry Entries

The following applies to Microsoft Internet Information Server 3.0 or earlier. If you are using IIS 4.0 or later, please refer to its documentation.

When the setup program installs ISAP on the Windows NT Server, it creates the following registry entries. If for some reason, you need to manually install ISAP, you need to create appropriate registry entries manually. All registry entries are installed under the following path:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC\Parameters
```

All entries described below specify the rest of the complete path relative to the path above.

Web Publishing Service Virtual Directories:

Path:	`\VirtualRoots`	Key:	`/Isap`	Value:	`<path to ISAP.DLL>,,4`
Path:	`\VirtualRoots`	Key:	`/Isapdoc`	Value:	`<documents directory>,,5`

Web Server Script Map:

Path:	`\ScriptMap`	Key:	`.xml`	Value:	`<path to ISAP.DLL>`

For example, if ISAP.DLL is located in the directory C:\InetPub\isap, the value for the first key above will be:

```
C:\InetPub\isap\isap.dll,,4
```